MODELING AND ANIMATION OF BRITTLE FRACTURE IN THREE DIMENSIONS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BİLKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Ayşe Küçükyılmaz

August, 2007

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Bülent Özgüç (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Veysi İşler

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Tolga Çapın

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

ABSTRACT

MODELING AND ANIMATION OF BRITTLE FRACTURE IN THREE DIMENSIONS

Ayşe Küçükyılmaz
M.S. in Computer Engineering
Supervisor: Prof. Dr. Bülent Özgüç
August, 2007

This thesis describes a system for simulating fracture in brittle objects. The system combines rigid body simulation methods with a constraint-based model to animate fracturing of arbitrary polyhedral shaped objects under impact. The objects are represented as sets of masses, where pairs of adjacent masses are connected by a distance-preserving linear constraint. The movement of the objects are normally realized by unconstrained rigid body dynamics. The fracture calculations are only done at discrete collision events. In case of an impact, the forces acting on the constraints are calculated. These forces determine how and where the object will break.

The problem with most of the existing fracture systems is that they only allow simulations to be done offline, either because the utilized techniques are computationally expensive or they require many small steps for accuracy. This work presents a near-real-time solution to the problem of brittle fracture and a graphical user interface to create realistic animations.

Keywords: Physically based modeling, real-time computer animation, brittle fracture, plastic deformation, crack formation, dynamics.

ÖZET

ÜÇ BOYUTTA KIRILMANIN MODELLENMESİ VE ANİMASYONU

Ayşe Küçükyılmaz
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Bülent Özgüç
Ağustos, 2007

Bu tezde kırılgan cisimlerin parçalanmasını benzeten bir sistem anlatılacaktır. Sistem, çok yüzlü üç boyutlu şekillerin etki altında kırılmasını canlandırmak için katı cisim benzetim yöntemlerini sınırlama bazlı bir model ile birleştirir. Cisimler, her iki komşu kütlenin uzaklık koruyan doğrusal bir sınırlama fonksiyonu ile birbirine bağlandığı kütle grupları olarak betimlenmiştir. Cisimlerin hareketi normal olarak serbest katı cisim dinamiğine uygun olarak gerçekleştirilir. Kırılma hesaplamaları sadece devamsız çarpışma durumlarında yapılır. Çarpışma durumunda sınırlamalara etki eden kuvvetler hesaplanır. Bu kuvvetler cismin nasıl ve nereden kırılacağını belirler.

Varolan kırılma sistemlerinin çoğundaki sorun, kullanılan tekniklerin hesaplama bazında pahalı olmasından ya da doğruluğu sağlamak adına çok sayıda ufak adıma gereksinim duymalarından ötürü benzetimlerin sadece çevrimdışı yapılmasına izin vermeleridir. Bu çalışma kırılgan cisimlerde parçalanma problemine yaklaşık gerçek zamanlı bir çözüm ve gerçekçi animasyonlar yaratmak için görsel bir arayüz sunmaktadır.

Anahtar sözcükler: Fizik tabanlı modelleme, gerçek zamanlı bilgisayar animasyonu, kırılma, deformasyon, çatlak oluşumu, dinamik.

Acknowledgement

I am deeply indebted to my supervisor Dr. Bülent Özgüç for his supervision, guidance, suggestions, and incredible patience throughout the development of this thesis. It was a great pleasure for me to have the chance of working with him.

I would also like to give special thanks to my thesis committee members Dr. Veysi İşler and Dr. Tolga Çapın for sparing their precious time for reading my thesis and their valuable comments.

Special thanks also go to Dr. Levent Onural, Selami Atlı, and Dilek Türk for having the patience with me when I failed to attend to my duties for the 3DTV NoE web site throughout the process of this thesis.

I would also like to thank Ozan Demir for his help in setting up a basis for this work and for having every confidence in me.

I would like to express my deepest thanks to the Togan family for their invaluable support. With you in my life, everything seemed so right. I would especially like to thank İnci Togan for giving me a dig when I was lost, and helping me survive the ordeal. Also thanks are due to Emre for cross-reading the thesis and his suggestions. Thanks for always listening to me and being a very good friend.

Besides, I want to thank Biter and Kıvanç for giving me encouragement and friendship, and to all the other people who have made Bilkent a very special place over all those years.

Last, but not the least, I would like to thank my family for their patience and sympathy, and for providing a loving environment for me.

This work is partially funded by EC within FP6 under Grant 511568 with the acronym 3DTV.

Contents

1 Introduction **1**

1.1 The System Architecture 3

1.2 Organization of the Thesis 3

2 Background **5**

3 Object Models **8**

4 Fracture Simulation **12**

4.1 Unconstrained Motion Calculations 14

4.2 Contact Calculations 17

4.2.1 Colliding Contact Calculations 20

4.2.2 Resting Contact Calculations 21

4.3 Fracture Calculations 23

4.3.1 Slow Fracture Simulation 24

4.3.2 Fast Fracture Simulation 27

4.4 Rendering 31

5 Dust Formation **32**

5.1 The Particle System 34

5.2 Rendering 36

6 Experimental Results **37**

6.1 Visual Results 37

6.2 Performance Analysis 40

7 Conclusion and Future Work **47**

A Colliding Contact Derivations **50**

B Resting Contact Derivations **55**

C The System at Work **61**

List of Figures

1.1 Overview of the animation generation process 3

3.1 A geometric object and its tetrahedral mesh, generated by NETGEN 9

3.2 Two tetrahedra and the corresponding lattice model. Adaped from [23] 10

3.3 The lattice and the tetrahedral mesh of an object rendered in white and blue respectively. 11

3.4 The cleaving effect 11

4.1 Six types of contacts according to the contacting features. In each condition contact points are marked red. 18

4.2 Three types of contacts according to the relative velocities 20

4.3 A comparison of the crack patterns generated by modifying the connection strengths uniformly (upper left image) and with the given algorithm (other images). 27

4.4 Effect of the fracture plane and radius r_{frac} on crack generation. Adapted from [12] 30

5.1 Dust formation upon impact 33

6.1 Ceramic bowl breaking upon falling to the ground (Animation generated by the slow algorithm) 38

6.2 Glass table breaking under the impact of a heavy ball (Animation generated by the fast algorithm) 39

6.3 A cube is broken with the first algorithm 41

6.4 A cube is broken with the second algorithm 42

6.5 A clay wall is broken with the slow algorithm 43

6.6 A clay wall is broken with the fast algorithm 44

6.7 Calculation times for the breaking cube animation (a) - slow algorithm (b) - fast algorithm 45

C.1 Main screen 62

C.2 File Menu 63

C.3 A sample scene 63

C.4 A selected object highlighted with red 64

C.5 Simulation Menu 65

C.6 Go to Frame Dialog 65

C.7 Object Properties Dialog 66

C.8 Apply Perlin Noise Dialog 66

C.9 Use Cleaving Planes Dialog 67

C.10 Algorithm choices within the Simulation menu 67

C.11 Window Menu 68

C.12 A Scene where Objects are Rendered as Meshes 68

C.13 A sample coloring scheme of the lattice after applying noise function on the object 69

List of Tables

6.1 Computation time of the fracture calculations versus impulse and tetrahedra counts for both algorithms 46

Chapter 1

Introduction

Fracture can be basically examined under two classes. We consider fracture brittle when only negligible plastic deformation takes place before separation. In other words, in brittle fracture the cracks form and propagate so easily that there is usually hardly enough time to watch the process. More fragile materials, that are shattered easily upon an impact applied on them, such as glass or ceramic, are materials prone to brittle fracture. On the other hand, in ductile fracture extensive plastic deformation takes place before fracture. Many pure metals, such as gold, copper and aluminum express high ductility.

Brittle fracture can be considered as being the worst type of fracture because of its irreversible effect on the material. Yet such effects are widely used in entertainment industry, mainly for the creation of expensive and difficult effects such as explosions, shattering and breaking of passive objects, and animation of natural phenomena. Such effects, when realized in the real world, may require a tremendous amount of money and might be overly destructive. Luckily, with the help of the computers, people can simulate these effects in a much cheaper way.

Modeling and simulation of fracture and deformation have been studied for over three decades in computer graphics [8]. Mostly, the simulations are done offline when physical precision is a concern. There are also real-time solutions which ignore several important material properties and sacrifice realism for the

sake of speed. Realistic animation of fracture is a difficult one. In order to generate a convincing animation, we need to understand the physical properties of the objects in a scene, rather than considering them as merely geometric shapes. These bodies should be thought of as real objects that have masses, elasticity, momentum, etc., and they display certain material properties such as the stress-strain relationship. Another difficulty of fracture animation is that the scenes change dynamically during the animation. The bodies are fragmented to create new bodies which are again subject to the same effects. Physically precise animations cannot be realized successfully by computation, since the real motions and the fragmentation of objects require an extensive amount of calculation. However, such great accuracy is not a requisite for animation purposes. By using physically based animation techniques, we can create realistic-looking shatters and breaks with much less effort, yet with as much visual precision as necessary.

In this thesis, we discuss a system implemented for generating computer animations of rigid objects that involve fracturing. The objects are represented by point masses connected with workless distance perserving constraints as proposed by Smith *et al.*[23]. This implementation combines continuum methods for simulating the fracturing of brittle objects with the rigid body simulation techniques in order to generate realistic-looking fracturing animations. Continuum methods supply the necessary stress-strain information for determining fracture effects within a body, but they are more expensive and mostly used in offline simulations. The physical motion of the objects are considerably fast because we neglect internal vibrations and treat the objects as rigid bodies between collision events. In a physical sense, a rigid body is an ideal solid in which deformation is neglected. Thus, between collisions the shapes of the rigid bodies remain intact.

The system implements two different techniques for the fracture process. The first technique computes the forces acting on the constraints within an object using Lagrange multipliers. The system solves a large sparse linear system at every time step, resulting with accurate but slow results. The second technique we used introduces some optimizations on computations and works in near-real-time.

1.1 The System Architecture

Creating a complete fracture animation is a multi step process. In our system, firstly, models are created for describing each object in the animation scene. The objects are initially represented by their polygonal surface descriptions. For the fracture process, a tetrahedral mesh and finally a lattice model of each object is constructed. After creation of these models, the animation steps are calculated according to the specified initial configurations of the objects. Finally, the calculated animation is rendered frame by frame, creating the final output of the program. Figure 1.1 gives a more detailed graphical overview of the animation generation process:

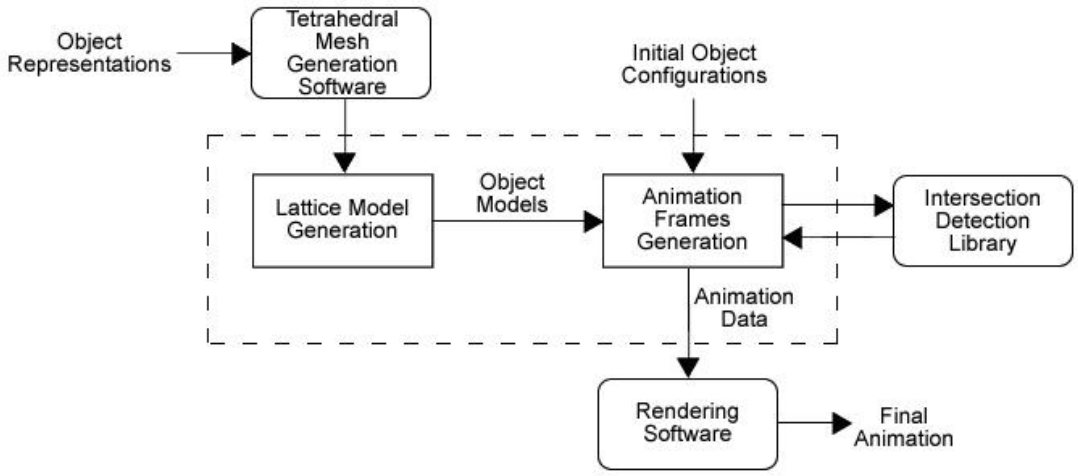

Figure 1.1: Overview of the animation generation process

1.2 Organization of the Thesis

The organization of this thesis is as follows: Chapter 2 provides a detailed review of the existing fracture and related deformation techniques in the literature. Chapter 3 explores the object model proposed by Smith, Witkin, and Baraff [23] that forms the basis for the simulation of the fracturing process we use. The process of generating each animation frame, which combines the rigid body and

the fracture simulation techniques, are described in Chapter 4. Chapter 5 provide the details of the dust formation feature of the system. Visual and numerical results of the system along with implementation details and a comparison between the optimized and the unoptimised system are provided in Chapter 6. Finally, Chapter 7 provides conclusions and suggestions for future work.

In the appendices A and B, you can find the contact calculation derivations skipped in the main text for the sake of clearness. Also detailed information on how the system works is presented in Appendix C.

Chapter 2

Background

In computer graphics community, modeling deformations has a long history. Since fracture process can be considered a deformation, many methods later used for the animation of brittle fracture is adapted from deformation studies. Hence in this section we will also examine some techniques developed for modeling deformations that act as a base for fracture studies, rather than just focusing on the area of brittle fracture.

The fracture process is hard to simulate using non-physical methods such as keyframing or by methods that use purely geometric approaches to model deformation where bodies are deformed by modifying some control points or shape parameters [8]. Even though the computations are very fast in such methods, since the system has no information about the manipulated bodies, the final animations depend mostly on the animator's expertise. As the objects get more complex, it becomes extremely hard to model the behavior.

In literature, there are very efficient non-physical methods that produce accurate results for generating crack patterns. These mainly map crack patterns to a surface or a volume (see [5, 10]). However, such methods are mainly used for generating crack patterns in solo images rather than running animations. Because voxelization or tetrahedralization of the objects are not necessary, these methods are not limited in resolution. Small or very thin crack patterns can be generated,

yet again, this process depends on the animator's talents.

Physically-based simulation is utilized in computer graphics for many applications. In the late eighties, Terzopoulos and Witkin introduced a hybrid model that represented a rigid body by its rigid and deformable components [27]. The rigid component handles the rigid-body motion while the elastic behavior is present only in the deformable displacement component. This approach aims solving the ill-conditioning of the discrete motion equations as the rigidity of the object increases. The computational cost, on the other hand, does not significantly improve with this method as intended.

Again, Terzopoulos and Fleischer generalized this work to encompass viscoelasticity, plasticity, and fracture in deformable bodies [26, 25]. They used forces that depend on the velocities of mesh points and spatial partial derivatives of the displacement function. Fracture is realized by breaking connections in the mesh. Although this work is not directly focused on fracture, the technique could be applied to animations such as tearing paper.

In 1991, Norton *et al.* [13] used a mass-spring system specifically for modeling fracture. In this model, the objects were subdivided into a set of equal sized cubes connected with springs. This elastic network proved to be cumbersome in case of the existance of large objects.

Later, in 1999, O'Brien and Hodgins [15] presented a work for crack initialization and propagation. Their method uses stress and strain tensors computed over a finite element method to determine the place where the cracks will initialize and in which direction they will propagate. Also they used a local re-meshing algorithm to correctly preserve the direction of the fracture by splitting the elements along the fracture boundaries to provide more realistic results. Later the techniques in this study are extended to handle ductile fracture as well [16, 14]. The results achieved by applying these techniques are strikingly good but the used continuum mechanics techniques are computationally demanding.

In 2000, Smith, Witkin and Baraff [23, 24] came up with a method that uses a system of point-masses connected by workless, distance preserving constraints to

model the object. The forces exerted by the rigid constraints are calculated using Lagrange multipliers. The simulation is realized as a result of solving a large, sparse, linear system using conjugate gradient method. The method presents a much faster solution than that of O'Brien *et al.* with still realistic outputs.

In 2001 Müller *et al.* [12] came up with the idea of using a hybrid approach similar to that of Terzopoulos *et al.* [27] for realizing deformations and fracture in real time. The difference is that their method is hybrid in time. The displacements, stresses and fracture bases are computed on a continuous model only at collision events. They use the Finite Element Method (FEM) as O'Brien *et al.* [15] do. However, they accelerate the core procedures of the FEM to work in near real time. The method uses the internal principal stress components provided by the FEM computation to determine fracture locations and orientations. In 2004, Müller *et al.* [11] extends the existing model to handle elastic and plastic deformations as well. Also, in this study, a method to animate fracture with a high resolution tetrahedral mesh is introduced. An object is represented by a low resolution volumetric mesh for the FEM simulation, and a high resolution surface mesh for rendering. The low resolution mesh provides enough information for accurate results whereas the high resolution surface mesh allows rendering to be done in the desired quality without loss of performance.

In 2005, Pauly *et al.* [17] proposed a different technique for simulating fracture. Their method, as opposed to the other techniques, does not work on a mesh for fracturing elastic and plastic materials. Instead, they use a highly dynamic surface and volume sampling method which is suppossed to solve stability problems in mesh-based techniques. System uses advancing crack fronts and fracture surfaces within a volume, where a body is composed of simulation nodes. When a crack is formed, the nodal shape functions are adapted. Complex fracture patterns can be achieved by altering the crack fronts via a small set of topological operations for splitting, merging, and terminating. This way, they achieve continuous crack patterns with detailed fracture surfaces.

Chapter 3

Object Models

The objects we use in the system are initially represented via either polygonal surface or Constructive Solid Geometry (CSG) representations. However such representations are not adequate for performing fracture calculations on. We adopted Smith *et al.*'s lattice model [23].

For generating the lattice model, initial object models are tetrahedralized. Since tetrahedral mesh generation is a separate research topic by itself, publicly available mesh generation software, NETGEN [21], is used for generating the tetrahedral meshes. NETGEN is an open-source automatic mesh generation tool for two and three dimensions. It comes as a stand-alone program with graphical user interface, and as a C++ library to be linked into another application. NETGEN accepts CSG and polygonal surface representations as input and creates their tetrahedral meshes. Also, applying a built-in uniform refinement algorithm can change the granularity of these meshes.

Once the tetrahedral mesh model is constructed, it is transformed into the final lattice representation. This representation is the Voronoi complement or dual of the solid object, which is basically a mesh of point masses -each representing a tetrahedron- connected to each other via workless distance-preserving constraints. Although it is similar to the simple spring-mass system, this model proves to be more advantageous for our purpose. Since we are dealing with brittle objects,

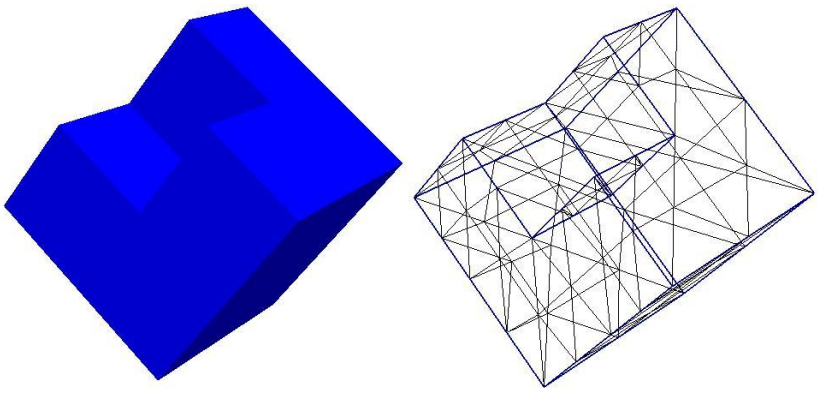

Figure 3.1: A geometric object and its tetrahedral mesh, generated by NETGEN

the springs should be stiff enough to prevent visible plastic deformations in a spring-mass system. For a perfectly brittle object, the springs are infinitely stiff. In the lattice model we use rigid constraints instead of springs, hence rather than calculating displacements, we calculate the forces that the constraints produce in response to an applied impulse. The point masses represent the micro-fragments of the object, and the constraints represent the physical strength of the bond between these micro fragments. In case the force applied on a bond due to some impact exceeds a limit, the bond is broken, realizing the fracture effect.

However it is crucial to understand that these constraints are not force vectors themselves. In a real-world material, the intermolecular bonds hold the elements together such that the total force acting on an element is zero, serving to ensure equilibrium. In such a case the sum of the vectors holding an element is always zero. Hence, if these constraints were such vectors, removing or changing the value of one of these would eventually change the neighboring vectors' values. In our case, the bonds only determine how close the material is to breaking at that specific point.

For generating the point masses and the constraints between them, the information from the tetrahedral meshes of our objects is used. For each tetrahedron in the mesh, a point mass is located at its center of gravity. The mass of each point is a function of the volume of the tetrahedron it represents, and the density of the material at that point. Also, for each pair of tetrahedra with a shared face, the corresponding point masses are connected with a rigid constraint, which has

strength proportional to the area of the shared face (Figure 3.2). By generating the lattice model in this manner, the geometry of the objects becomes the effective factor in determining its breaking behavior.

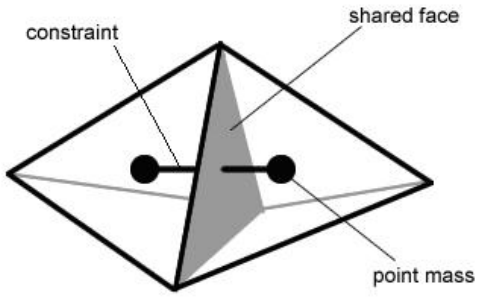

Figure 3.2: Two tetrahedra and the corresponding lattice model. Adaped from [23]

Figure 3.3 displays the tetrahedral mesh and lattice of a simple object. The tetrahedral mesh is rendered in blue. The lattice as drawn in this figure consists of the connections within the model. In other words, the lines rendered in white are rigid constraints that connect the masses. Upon closer inspection, the reader can observe that each tetrahedron is connected with its neighbors via white lines.

Some heuristics can also be applied after this step to achieve some user control and flexibility on the fracturing behavior of the objects. Cleaving planes are used for systematically reducing or increasing the connection strengths along a cross section of the objects. This way the user can define areas that are required to break or remain intact. Figure 3.4 illustrates the effect of cleaving on an object. The frame is generated by cleaving the rectangular block and letting it fall to the ground. The block is subjected to four cleaving operations, where the regions between parallel planes are weakened and other regions are strengthened for a more dramatic result. When the block hits the ground, fracture takes place in the weak regions, which eventually make up a cross and a plus sign on the block. Three-dimensional noise and turbulance functions [18, 19] also provide a way to procedurally change the connection strength of the objects to achieve different fracturing behavior under the same initial conditions.

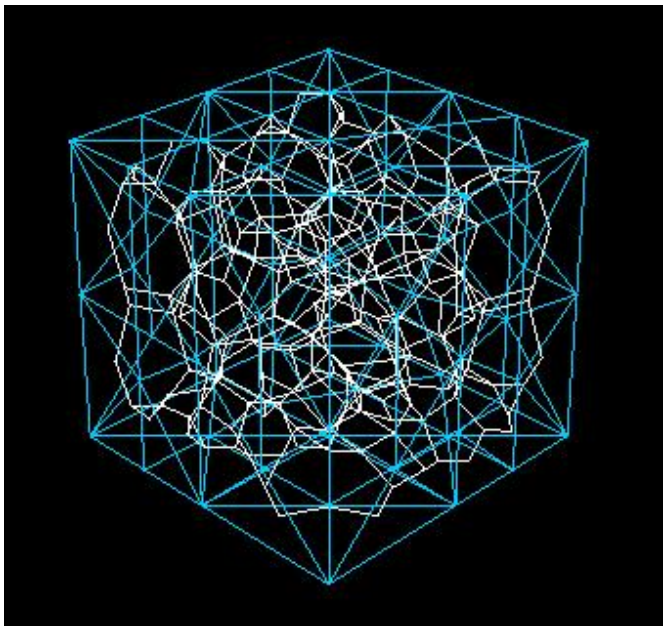

Figure 3.3: The lattice and the tetrahedral mesh of an object rendered in white and blue respectively.

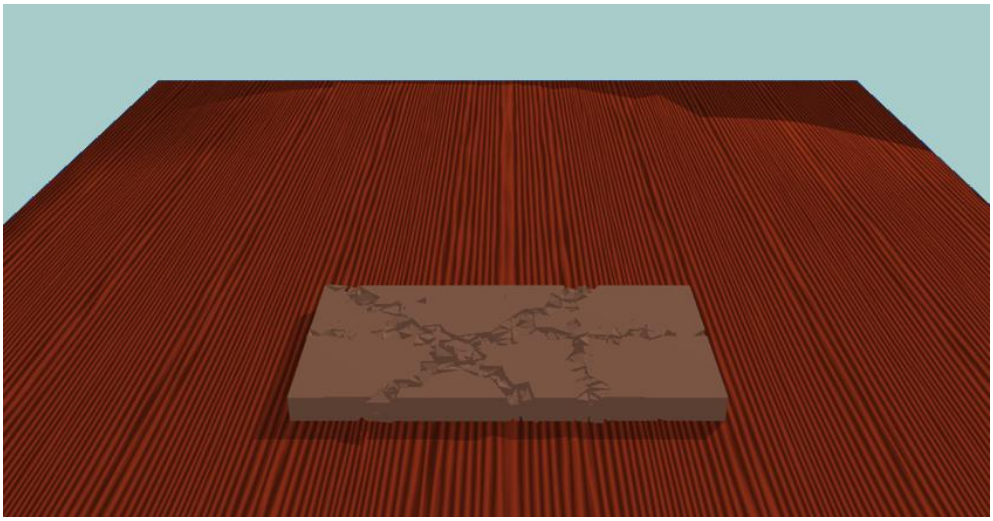

Figure 3.4: The cleaving effect

Chapter 4

Fracture Simulation

Our system is mainly a tool for manipulating objects in a 3D scene. It is a physics simulator in which objects can be subject to fracture. The fracture can be the result of a direct impact on the object as well as the indirect effect of an artificial force field. As mentioned earlier, the fracture calculations are only done at collision instants. The bodies act as unconstrained rigid bodies when there are no impacts. From now on, the notation used by Witkin *et al.* [30] will be used to denote derivatives, where a single dot is used for the first order time derivative and two dots denote second order time derivative.

The simulator works by generating an animation frame per time step. For generating the animation frames, the motion paths of the objects in the scene are calculated, and their updated positions and orientations are determined for each frame. In the case of a contact between two objects, the motion paths of the objects are updated and fracture calculations are performed. If these calculations result in the shattering of the object, the resulting shards are modeled as new objects and they are included in the animation calculations. This process is repeated until all the desired frames of animation are generated.

For updating the positions and orientations of the objects at each frame, a technique called bisection is applied. As it can clearly be seen from the pseudo-code description below, this technique divides a single interval between two frames

into smaller sub-intervals, ensuring that the objects are in valid configurations at the end of each sub-interval.

Algorithm: Generating the Animation Frames

```
GENERATE_FRAMES(numFrames, frameDuration)
 1  currentFrame ← 1
 2  while currentFrame < numFrames
 3     remainingDuration ← frameDuration
 4     updateDuration ← frameDuration
 5     while remainingDuration > 0
 6           UPDATE_OBJECTS(updateDuration)
 7           if no objects intersect
 8                 remainingDuration ← remainingDuration − updateDuration
 9                 updateDuration ← remainingDuration
10                else updateDuration ← updateDuration/2
11     PROCESS_CONTACTS
12     PROCESS_FRACTURE
13     if dust creation is selected
14          create particle systems where necessary
```

Determining whether the objects are in valid configurations at the end of each sub-interval requires intersection tests between each pair of objects. For performing these tests, a publicly available software library, FreeSOLID [28], is used. After creating the object models, each object is also given to the library by describing their shapes and initial configurations. During the generation of the animation frames, the configuration changes of the objects are also reflected to FreeSOLID and the intersection tests are performed via the functionality provided by the library.

After getting a valid configuration, calculations are performed for handling the contacts between the objects as well as determining the fracturing of the objects. In the remainder of this chapter, details of the motion, contact and

fracture calculations are discussed. The derivations in sections 4.1 and 4.2 are based on the course notes prepared by Baraff [3].

4.1 Unconstrained Motion Calculations

Unconstrained motion means that the motion of the body is not restricted by any constraints so that any position in the space is valid for the object. Although, obviously this is not the case in real life and in fracture animation, these calculations form the basis of animation calculations since they handle all the situations that do not involve collisions.

We define the state of an object in the space by its state vector $X(t)$ as:

$$X(t) = \begin{bmatrix} x(t) \\ Q(t) \\ P(t) \\ L(t) \end{bmatrix} . \tag{4.1}$$

Here, $x(t)$ is a 3-vector that describes the position of the center of mass of the object, $Q(t)$ is a quaternion that describes the orientation of the object around its center of mass, $P(t)$ and $L(t)$ are 3-vectors that respectively describe the linear and angular momentums of the object.

The linear momentum of an object $P(t)$ is defined as:

$$P(t) = mv(t) . \tag{4.2}$$

Hence, the linear velocity $v(t)$ of an object is

$$v(t) = \frac{P(t)}{m} . \tag{4.3}$$

Since $x(t)$ is the position of the center of mass of the object, its time derivative is simply the linear velocity of the object.

$$\dot{x}(t) = v(t) \ . \tag{4.4}$$

Similarly, the angular momentum of an object $L(t)$ is defined as:

$$L(t) = l(t)w(t) \ , \tag{4.5}$$

where, $l(t)$ is the rank-two inertia tensor for an object consisting of n point masses, each with mass m_i, and position vector r_i relative to the position of the center of mass of the object:

$$l(t) = \sum_{i=0}^{n} \begin{bmatrix} m_i({r'_{iy}}^2 + {r'_{iz}}^2) & -m_i r'_{ix} r'_{iy} & -m_i r'_{ix} r'_{iz} \\ -m_i r'_{ix} r'_{iy} & m_i({r'_{iy}}^2 + {r'_{iz}}^2) & -m_i r'_{iy} r'_{iz} \\ -m_i r'_{ix} r'_{iz} & -m_i r'_{iy} r'_{iz} & m_i({r'_{iy}}^2 + {r'_{iz}}^2) \end{bmatrix} \ . \tag{4.6}$$

Similar to the linear velocity case, the angular velocity of the object can be defined as:

$$w(t) = l^{-1}(t)L(t) \ . \tag{4.7}$$

If the angular velocity of an object is $w(t)$, this means that the object is rotating about the axis $w(t)$ with a velocity of magnitude $|w(t)|$. Thus, the orientation of the object after rotating with the angular velocity of $w(t)$ for a period of time $\Delta t = t - t_0$ can be represented with the quaternion:

$$Q(t) = \left[\cos\left(\frac{|w(t)|\,\Delta\ t}{2}\right), \sin\left(\frac{|w(t)|\,\Delta\ t}{w}\right)\frac{w(t)}{|w(t)|},\right] Q(t_0) \ . \tag{4.8}$$

Taking $t = t_0$ and differentiating with respect to time, we get the formula for $\dot{Q}(t)$:

$$\dot{Q}(t) = \left[0, \frac{w(t)}{2}\right] Q(t_0) \ . \tag{4.9}$$

The linear acceleration of the object can be described by taking the time derivative of equation 4.3:

$$\dot{v}(t) = \frac{\dot{P}(t)}{m} \ . \tag{4.10}$$

Reorganizing and applying Newton's second law of motion we get

$$\dot{P}(t) = m\dot{v}(t) = F(t) \ . \tag{4.11}$$

Similarly, the time derivative of the angular momentum of the object can be defined by taking the time derivative of equation 4.5:

$$\dot{L}(t) = \dot{I}(t)w(t) + I(t)\dot{w}(t) = \tau(t) \ , \tag{4.12}$$

where $\tau(t)$ is the total torque acting on the object.

By combining equations 4.4, 4.9, 4.11 and 4.12, the time derivative of the state vector can be constructed as:

$$\dot{X}(t) = \begin{bmatrix} v(t) \\ \dot{Q}(t) \\ F(t) \\ \tau(t) \end{bmatrix} \ , \tag{4.13}$$

where $v(t)$ is the linear velocity of the object, $F(t)$ is the total force acting on the object and $\tau(t)$ is the total torque acting on the object. The state of an object at time t is defined by the ordinary differential equation (ODE):

$$\frac{d}{dt}X(t) = \dot{X}(t) \ . \tag{4.14}$$

For calculating the motion of an object, this ODE must be integrated for the duration of each animation frame. At the beginning of the animation, the initial values for four components of the state vector are given as the initial configuration of the object, and during the course of the animation the state vector is updated at each motion calculation. This way, the motion calculation becomes an initial value problem from which the value of $X(t)$ can be computed for any value of t. An ODE integrator using the Fourth-Order Runge-Kutta method with adaptive step sizing is implemented to solve this initial value problem and calculate the motion of the objects (see [20]). By applying an adaptive step sizing algorithm, an upper bound for error is maintained in motion calculations.

4.2 Contact Calculations

So far, the motion calculations are performed with the assumption that the movement of the object is unconstrained. However this assumption is not valid when two objects come into contact with each other since motion of real world objects is constrained with impenetrability constraints. In other words, in real world no two objects can have overlapping volumes. Thus, in order to be able to realistically simulate the motion of the objects in the animation scene, the impenetrability constraints must be enforced [2].

A constraint can be defined by a function which takes certain values when it is satisfied. The impenetrability constraints can be defined as inequality-constrained constraints such that for the constraint function $C((x(t))$, we should have:

$$C(x(t)) \geq 0 \ , \tag{4.15}$$

for all valid positions $x(t)$ of the object.

Whenever the impenetrability constraints are satisfied, we consider the motion of the object as unconstrained and perform the calculations described in the previous section to determine it. In the case when an impenetrability constraint is unsatisfied, the last point in time when the constraint was satisfied is found

using the bisection technique described earlier and necessary actions are taken in order to make the object satisfy the constraints in the future.

If the impenetrability constraint for a pair of objects is not satisfied at a given time t, this means that these objects have overlapping volumes, i.e. they are intersecting. In such a case the last point in time when the constraint was satisfied is the time when the objects were in contact. So, by applying appropriate response on the objects when they are in contact, the impenetrability constraints can be enforced throughout the animation.

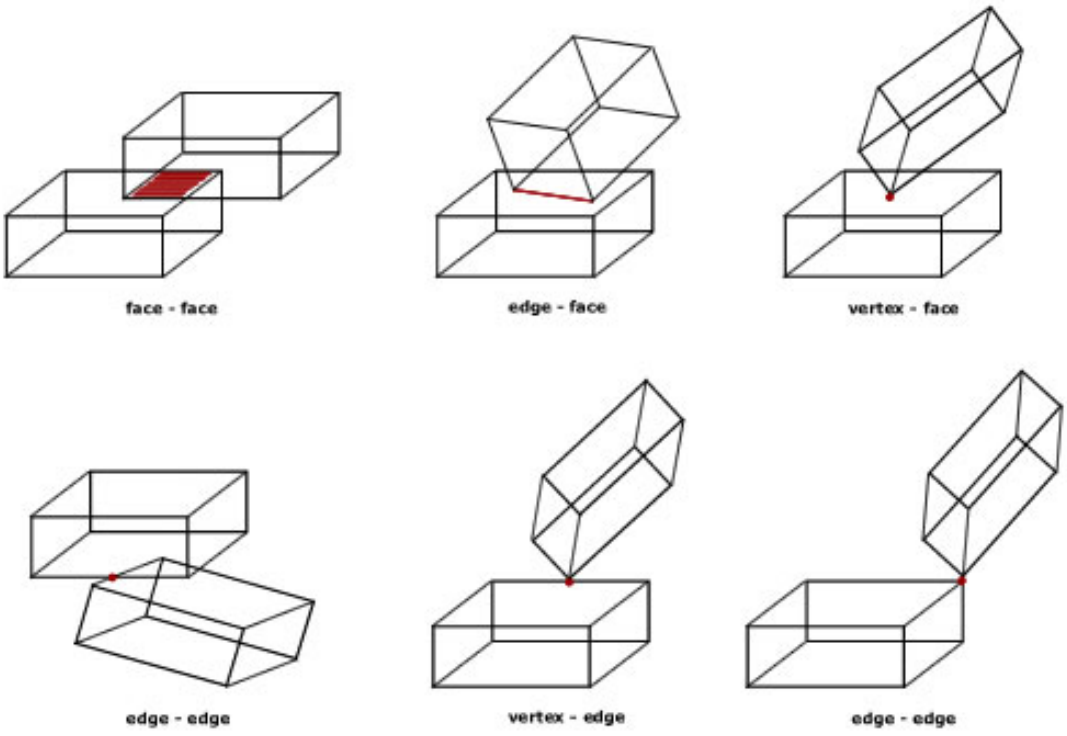

Figure 4.1: Six types of contacts according to the contacting features. In each condition contact points are marked red.

A contact between two objects is defined by the contact normal and a contact point extracted from contacting features of the two objects. In the case of multiple contact points between two objects, each contact point is considered as a separate contact. In terms of contacting features, contacts can be categorized into six

groups (see Figure 4.1). Two of these contact types, vertex-edge and vertex-vertex, are considered degenerate in the sense that a contact normal cannot be determined for them. Since the probability of a degenerate contact occurring between two objects is very low, they are discarded while handling the contacts [1]. For the remaining four contact types the contact points and the corresponding contact normals are determined as follows:

- **Face-Face:** The contact normal is the surface normal of one of the contacting faces. The contact points are the vertices of the polygon that define the intersection of the faces.
- **Edge-Face:** The contact normal is the surface normal of the contacting face. The contact points are the two endpoints of the edge segment that intersect with the face.
- **Vertex-Face:** The contact normal is the surface normal of the face. The only contact point is the contacting vertex.
- **Edge-Edge:** In this case it is assumed that the contacting edges are not collinear. So, the contact normal is the normal of the plane defined by the lines that go through the two contacting edges. The only contact point is the intersection point of the two edges.

After the contact points and the corresponding contact normals of a contact are determined, the appropriate response for the contact is determined by looking at the projection of the relative velocity of the objects at the contact points over the contact normal. Figure 4.2 shows the three possible cases which are handled differently.

If the projected relative velocity is greater than zero, it means that the contacting objects are actually separating from each other, thus no response is necessary to enforce the impenetrability constraint. If the projected relative velocity is zero, it means that the contacting objects are moving together. This type of contact is called a resting contact and it is handled by calculating contact forces that act on the objects at the contact points. If the projected relative velocity is less than

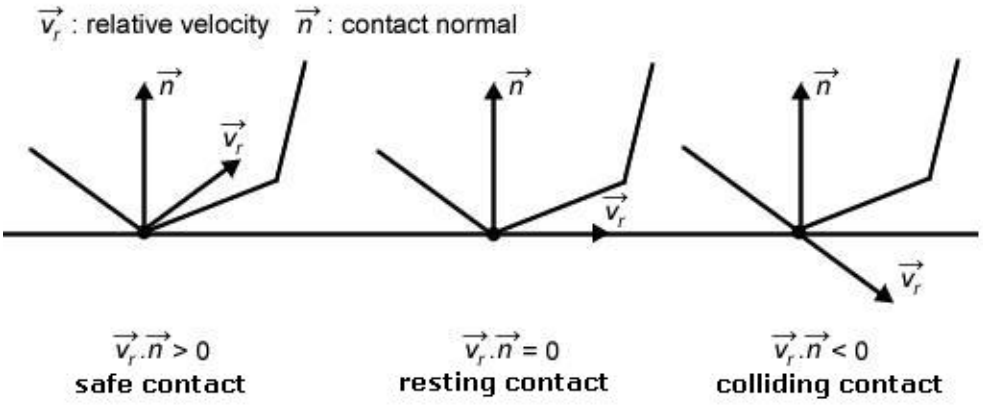

Figure 4.2: Three types of contacts according to the relative velocities

zero, it means that the contacting points are moving towards each other. This type of contact is called a colliding contact and it is handled by calculating contact impulses that act on the objects at the contact points. In the following two subsections the details of the colliding contact and resting contact calculations will be discussed.

4.2.1 Colliding Contact Calculations

Colliding contact is the type of contact at which the contacting objects are moving towards each other. Mathematically, the quantity v_{rel} can be defined as:

$$v_{rel} = \hat{n}(t_c)(\dot{p_a}(t_c) - \dot{p_b}(t_c)) \ , \tag{4.16}$$

where, $\dot{p_a}(t)$ and $\dot{p_b}(t)$ are the velocities of the contact points on the contacting objects, $\hat{n}(t)$ is the unit contact normal vector and t_c is the time of the contact. The quantity v_{rel} gives the component of the relative velocity $\dot{p_a}(t_c) - \dot{p_b}(t_c)$ in the $\hat{n}(t_c)$ direction. In case v_{rel} is positive, the bodies are safely moving away from each other. If v_{rel} is negative then we have a colliding contact. v_{rel} being zero implies that the bodies are resting on each other and this case is explained in the next section.

In real life an elastic collision is a process in which the colliding objects remain

in contact for some duration of time. During this duration contact forces, which are equal in magnitude and reverse in direction, act on the objects. However, since all the objects in the animation scene are brittle objects, the collisions can safely be reduced to instantaneous events without reducing the quality of the resulting animation. By reducing the contact duration to an instance, it is also necessary to reduce the effect of the contact forces over the duration of the collision to a single instance. The vector I, which is the impulse that acts on objects that are in contact can be defined as:

$$I = \int F(t)dt \ , \tag{4.17}$$

where, $F(t)$ is the vector function that defines the contact force acting on the object during the course of the contact. The required change in the velocity of the object due to this collision can be achieved by applying the changes $\Delta P(t)$ and $\Delta L(t)$ to the linear and angular momentums of the object respectively:

$$\Delta P(t_c) = I \tag{4.18}$$

$$\Delta L(t_c) = (p - x(t_c)) \times I \ . \tag{4.19}$$

Here, p is the contact point, $x(t)$ is the position of the center of mass of the object at time t, and t_c is the time of the collision. Detailed derivations of these formulas, and details on how to calculate the impulse vector I can be found in Appendix A.

4.2.2 Resting Contact Calculations

In the case of the resting contact the contacting objects are neither moving towards each other nor moving apart at the contact point. However the impenetrability constraint can still be violated if the contacting objects are accelerating towards each other. In this case contact forces must be calculated and applied on the objects in order to prevent them from accelerating into each other.

The time derivative of v_{rel}, which is the projection of the relative acceleration of the objects over the contact normal at the contact point, can be defined as:

$$\dot{v}_{rel} = a_{rel} = \hat{n}(t_c) \cdot (\ddot{p_a}(t_c) - \ddot{p_b}(t_c)) + 2\dot{\hat{n}}(t_c) \cdot (\dot{p_a}(t_c) - \dot{p_b}(t_c)) \ , \tag{4.20}$$

where $\ddot{p_a}(t)$ and $\ddot{p_b}(t)$ are the accelerations and $\dot{p_a}(t)$ and $\dot{p_b}(t)$ are the velocities of the contact points on the contacting objects, $\hat{n}(t)$ is the unit contact normal vector and t_c is the time of contact.

Besides requiring a_{rel} to be nonnegative, two other conditions must be satisfied while calculating the contact forces. Firstly, the contact forces should never be attractive. Secondly, the contact forces must remain as long as the corresponding contact remains and no more. By combining these conditions together, we can formulate the problem of finding the contact forces as a quadratic programming (QP) problem as follows:

$$\min f^T(Af + b) \text{ subject to } \begin{cases} (Af + b) \geq 0 \\ f \geq 0 \end{cases} . \tag{4.21}$$

Here $(Af + b)$ is the concatenation of all the a_{rel} values for all of the resting contacts and f is the concatenated vector of contact forces that are required for enforcing the impenetrability constraints. The concatenated vector is separated into its force dependant and force independent parts in order to be able to formulate it as a QP problem. The details of formulating the QP problem can be found in Appendix B.

After the QP problem is formulated, it is given to the OOQP library [6, 7], a publicly available QP problem solver, and the resulting contact forces are applied to the objects.

4.3 Fracture Calculations

As mentioned earlier, our system uses two different techniques for realizing fracture. In both techniques the simulation of the fracturing process makes use of the lattice model representation of the objects. The crack initialization is invoked due to some external force applied to a point on the outer surface or in the inner region of the object. Obviously, the forces can be either artificial, explicitly defined to initiate a fracture, or impulses due to colliding contacts, calculated while enforcing the impenetrability constraints. Upon the application of such a force, in response, constraint forces are calculated and the constraints are modified accordingly. What differs in the two approaches implemented for this thesis is how we modify the constraints in the model. Yet, regardless of how the modification is done, we enforce the distance-preserving constraints on the lattice of particles. In case the constraint force for a connection is greater than the current constraint strength, that constraint is removed. Otherwise the existing constraint strength is weakened by the amount of the constraint force applied on it. Any constraint for which the resulting constraint strength is weaker than a predefined constraint force threshold is removed.

In the next two sections, we will examine two different algorithms used for animating the fracture effects. The first algorithm explained in Section 4.3.1 is proposed by Smith *et al.* [23, 24]. This algorithm computes fracture as a local event, determining the force acting on the tetrahedra using Lagrange multipliers. Whenever there is an impact on the body such forces are computed and bonds are broken to initiate fracture. Applying the forces in a stepwise manner helps obtaining crack propagation. However no global information is valid for material properties. Hence the cracks seem to occur at collision regions only. Also it is very hard to simulate different materials with this algorithm. The terms *first algorithm*, *slow algorithm*, *slow solution*, *first solution*, and *slow fracture simulation* will be interchangeably used for citing this algorithm.

The second algorithm explained in Section 4.3.2 is proposed by O'Brien *et al.* [15], and then optimized by Müller *et al.* [12] to run in near-real-time. This method makes use of stress tensors within an object when calculating the

fracture. In fact, at each tetrahedra of the model stress is hidden. Upon an impulse this stress is exposed and in our case this stress is expressed as fracture. This formulation assumes the fracture will start at the tetrahedra where the stress is maximum. When such a tetrahedra is found, a collision plane is constructed to form the cracks. This algorithm represents the body globally. It is expected to see large cracks where the body is prone to failing, instead of having all cracks near impact region. The terms *second algorithm*, *fast algorithm*, *near-real-time algorithm second solution*, *fast solution*, *near-real-time solution*, and *fast fracture simulation* will be interchangeably used when talking about this algorithm.

4.3.1 Slow Fracture Simulation

For calculating the constraint forces that act on the system of particles of the object, the positions of the particles are placed in a vector named q, such that, for an n particle system, q is a $3n \times 1$ vector defined as:

$$q = \begin{bmatrix} q_1 \\ \vdots \\ q_n \end{bmatrix} . \tag{4.22}$$

A mass matrix M is defined in such a way that it holds the particles' masses on the main diagonal, and 0's elsewhere. So a mass matrix for n particles in 3D is a $3n \times 3n$ matrix with diagonal elements $\{m_1, m_1, m_1, m_2, m_2, m_2, ..., m_n, m_n, m_n\}$.

Finally, a global force vector Q is obtained by joining the forces on all particles, just as we did for the positions. From Newton's Second Law of Motion, the global equation on the particle system is as follows:

$$\ddot{q} = M^{-1}Q \ , \tag{4.23}$$

where M^{-1} is the inverse of the mass matrix, M.

A similar global notation will be used for the set of constraints: Concatenating all scalar constraint functions form the vector function $C(q)$. In 3D, for n particles

subject to m constraints, this constraint function has an input of a $3n \times 1$ vector, and an output of an $m \times 1$ vector. In our case, this constraint function consists of the scalar distance-preserving constraints in the form:

$$C_i(p_a, p_b) = \|p_a - p_b\| - d_i \ , \tag{4.24}$$

where p_a and p_b are the positions of two particles connected to constraint i, and d_i is the distance between the particles that needs to be preserved.

Assuming initial positions and velocities are legal, we try to come up with a feasible constraint force vector $\hat{Q}$ such that the distance preserving constraints are held. In other words, for the initial conditions that satisfy $C(q) = \dot{C}(q) = 0$, we are trying to find the constraint force vector $\hat{Q}$, such that, when added to Q, guarantees $\ddot{C}(q) = 0$. In order not to break the balance of the system, it has to be assured that no work is done by the constraint forces in system, for all valid position vectors:

$$\hat{Q}.\dot{q} = 0, \quad \forall \dot{q} | J\dot{q} = 0 \ , \tag{4.25}$$

All vectors that satisfy this requirement can be written as

$$\hat{Q} = J^T \lambda \ , \tag{4.26}$$

where J is the Jacobian of C, and is equal to $\frac{\partial C}{\partial q}$. Here, λ is a vector with the same dimensions as C. The components of λ are known as Lagrange multipliers and they tell how much of each constraint gradient is mixed into the constraint force. From 4.26:

$$JM^{-1}J^T\lambda = -\dot{J}\dot{q} - JM^{-1}Q \ . \tag{4.27}$$

Note that the above formula is a system of linear equations of the form $Ax = b$ where A is a matrix and x and b are vectors. By calculating the λ vector from equation 4.27 and placing it in equation 4.26, the constraint force vector $\hat{Q}$, which satisfies the given rigidity constraints can be calculated. To ensure that the λ vector is a physically realizable solution for the system, the conjugate gradient method [22], which gives the minimum norm solution of the system, is used since the minimum norm solution of the system is also the physically realizable one.

Even though the constraint force vector $\hat{Q}$ could be found in a single step, it produces better results to do it in multiple steps, transmitting forces among the particles at each step before they are removed, given that they transmit no more force than the breaking stress would tolerate. Thus, each $\hat{Q}$ is used as an input to the next iteration.

Solving the problem in multiple steps, the impact force is increased gradually at each step. This way, as opposed to creating a single impulse, a more realistic impulse history is created.

4.3.1.1 Modifying the Connection Strengths

Once a crack is invoked at some point of the model, due to some external or internal force, the connection strengths are modified procedurally. In real world, when a brittle object is breaking, the energy required for starting a new crack is much higher than propagating an already existing crack. Thus, to imitate this behavior, while removing a newly broken constraint, the neighboring constraints, which are adjacent to the faces of the newly broken constraint, are weakened. This way, the relative probability of growing of an existing crack to the initiation of a new one is increased.

Obviously, the connections that are close to the crack region will be affected more than the connections that are far away from it. The strengths are modified gradually. However, weakening the connection strengths uniformly produces cracks that are visually artificial. Hence, in order to introduce a randomness into the crack pattern, some connections are made weaker than the others. These connections, and the amount of weakening are selected randomly. This operation introduces no performance loss, yet it is very successful in generating crack patterns. Moreover, even though two geometrically same objects are broken under the same conditions, the system produces distinct final crack patterns and formation of longer cracks is achieved.

Fig. 4.3 compares the effect of modifying the connections with and without the given technique. The object in upper left image is broken with the original

algorithm, while the objects shown in the other three images are broken with our modified one. It is easily observable how the crack patterns change every time the algorithm is run. In addition, with the technique used here, not only a successful randomization in cracks is achieved, but also the cracks formed after the fracture are longer.

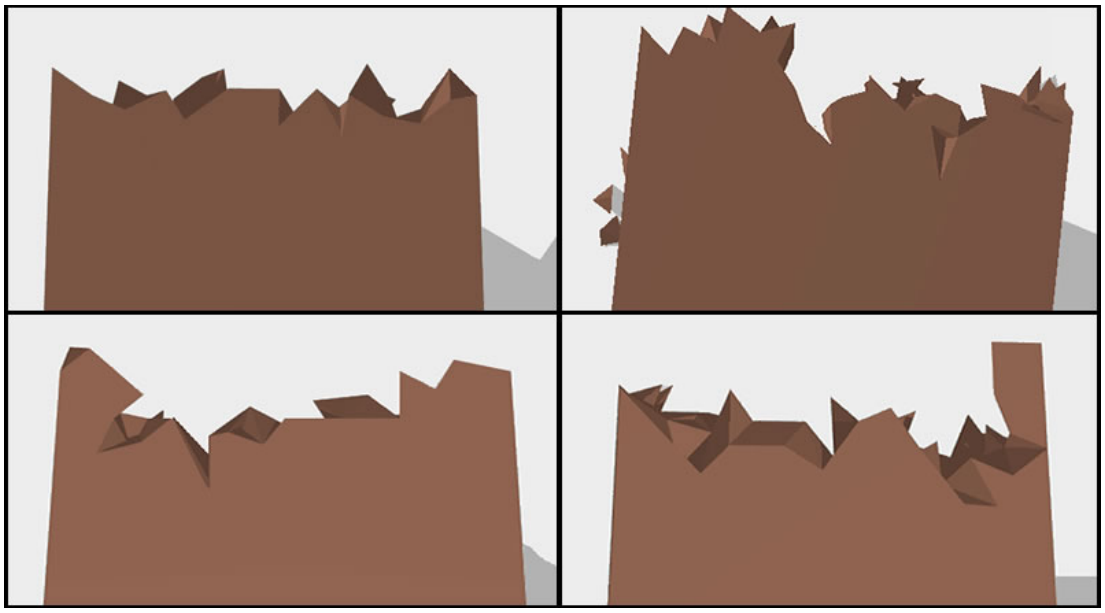

Figure 4.3: A comparison of the crack patterns generated by modifying the connection strengths uniformly (upper left image) and with the given algorithm (other images).

4.3.2 Fast Fracture Simulation

Our near-real-time solution makes use of the stress supported by a body when handling fracture as proposed by O'Brien *et al.* [15] and then improved by Müller *et al.* [12] with slight modifications.

Stress is a measure of force acting on per unit area within a body. Cauchy's principle states that, within a material the forces imposed by a closed volume are in equilibrium with the forces imposed on the closed volume by the remainder of the body [29]. Hence we are trying to find a set of forces imposed on each tetrahedra in response to a force acting on the body.

The stress at a point can be found by considering a small element of the body with area ΔA, perpendicular to which a force ΔF is applied. By making the element infinitely small, the scalar stress σ can be found:

$$\sigma = \lim_{\Delta A \to 0} \frac{\Delta F}{\Delta A} = \frac{dF}{dA} . \tag{4.28}$$

Stress can be described by a second-order tensor since the behaviour of the body is independent of the coordinate systems the body is located on. In 3D, the internal force acting on area ΔA of a plane can be partitioned into three components: one normal to the plane and two parallel to the plane. Divided by ΔA, the normal component gives the **normal stress**, denoted by σ; and the parallel components give the **shear stresses**, denoted by τ. The Cauchy stress tensor is a 3×3 matrix.

$$\sigma = \begin{bmatrix} \sigma_x & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_y & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \sigma_z \end{bmatrix} . \tag{4.29}$$

Here, $\sigma_i, i \in \{x, y, z\}$ denotes the normal stress and τ_{ij}, with $i, j \in \{x, y, z\}$ and $i \neq j$, values are shear stresses as mentioned earlier. In case of equilibrium $\tau_{xy} = \tau_{yx}$, $\tau_{xz} = \tau_{zx}$, and $\tau_{yz} = \tau_{zy}$. Hence the tensor is also symmetric. This matrix has 3 real eigenvalues, which denote the principal stresses. The related eigenvectors denote the principal stress directions. A positive eigenvalue stands for tensile stress while a negative one indicates compressive stress.

What Müller *et al.* [12] do for determining the breaking behaviour object is to perform an eigenvalue decomposition for each tetrahedron in the object model. This way, they compute the stress at each atomic unit of the material. Let λ_{max} be the largest eigenvalue calculated for the whole body. If λ_{max} is greater than a given threshold, fracture takes place. The tetrahedron with the greatest eigenvalue is where the crack initialization will begin. This is a realistic approach since in real world fracture is initiated to release the greatest energy in response to an external/internal effect.

A fracture plane perpendicular to the eigenvector of λ_{max} is obtained upon an impact. and within a radius r_{frac}, tetrahedra are split if their center of masses fall on opposite sides of the fracture plane (see Figure 4.4). r_{frac} is determined according to the material properties and the stress magnitude.

Only the tetrahedra within a radius r_{frac} from the tetrahedron under tensile stress -the one with the largest eigenvalue- are examined. The constraint strengths between neighboring tetrahedra, the center of masses of which fall on opposite sides of the fracture plane are updated according to the stress acting on them.

Our approach differs slightly from this approach, not by methodology, but by the number of affected tetrahedra and the calculation of the stress tensor. We, instead of calculating the eigenvalues of all tetrahedra, do a localized analysis. Our system works on the assumption that the largest stress will be close to the impact point. Experiments showed that instead of calculating the eigenvalues for all tetrahedra, only considering the ones close enough to the impact point proves to give satisfying results. In fact, in most cases, these tetrahedra are most prone to stress within a body. This way, we radically decrease the number of computations.

Also for the stress tensor we do a simplified calculation. We ignore the effects of the shear stresses, this way, the eigenvalues are equal to the normal stress values, while unit vectors are the eigenvectors. For the diagonal elements σ_i's, we use the x, y, z components of an effective force f_{eff} acting on a tetrahedron. Firstly the average of all contacts that are fairly close to each other are used to form the vector f_{avg}. The closeness is determined by thresholding the distance between two impact points. Then the effective force is calculated as:

$$f_{eff} = (f_{avg}/(r_t)^2) * vol(t) \ , \tag{4.30}$$

where r_t is the distance between the tetrahedron's center of mass and f_{eff}'s application point; and $vol(t)$ is the volume of the corresponding tetrahedron t. For a tetrahedron with vertex positions p_1, p_2, p_3, and p_4, the volume is

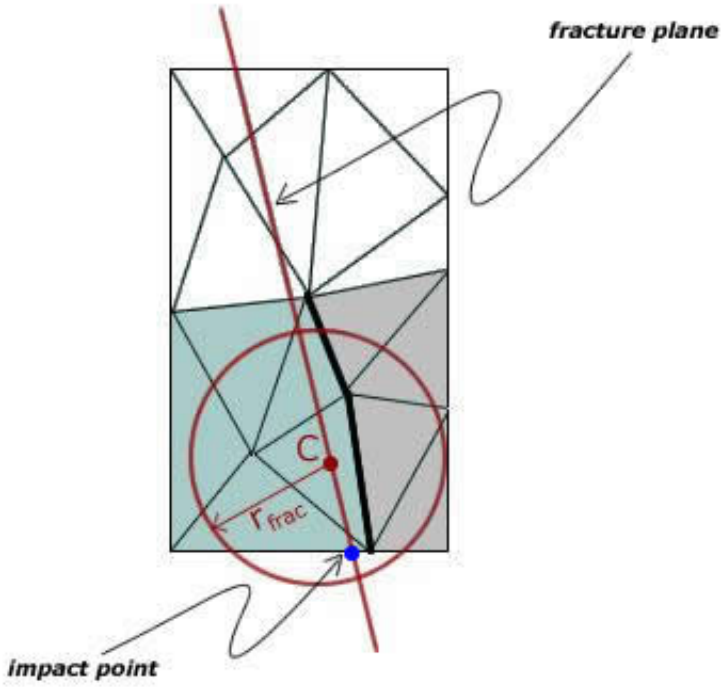

Figure 4.4: Effect of the fracture plane and radius r_{frac} on crack generation. Adapted from [12]

$$vol = \frac{1}{6}\left[(p_2 - p_1) \times (p_3 - p_1)\right] \cdot (p_4 - p_1) \ . \tag{4.31}$$

Another modification to Müller's method is done when modifying the related connection strengths. Müller removes all connections within a given radius through the fracture plane if the largest eigenvalue exceeds a given limit. However, we chose to modify the connection strengths within the boundary by the relative stress acting between two tetrahedra. Changing the connection strengths, again, is done as a multi-step process to enhance the reality of the resulting animations.

4.4 Rendering

The last step of animation generation process is rendering the animation scenes frame by frame, which creates a realistic looking output. After calculating each frame of the animation, the program outputs a file that the rendering software can read and render, which includes only the object data on scene. By changing the global effects, such as lighting, camera, etc, several animations can be prepared for a single fracture.

Persistence of Vision Raytracer (POV-Ray) software, v3.6 [4] is used for realizing this step. The software has a scripting language, namely POV-Ray 3 Scene Description Language, which allows an easy way to describe a scene in plain ASCII text format. The language, similar to many widely used programming languages, consists of identifiers, reserved keywords, floating-point expressions, strings, special symbols, and comments. It also includes predefined structures for several shapes, such as spheres, planes etc. In addition there are methods for modifying the camera location and angle, as well as the illumination of the scene. POV-Ray allows applying texture mapping or photon mapping to the objects; adding interiors to them, i.e. to give a feeling of glass; and adding atmospheric effects such as fog, to the scene. In addition, the light sources, which vary from point light sources to spotlights, can be used extensively, where a scene can contain multiple light sources.

Since the objects in the implemented solution are described as tetrahedral meshes, the Mesh2 object of POV-Ray, which is a representation of a mesh, is used. A Mesh2 object is given the coordinates of each vertex, and information on which vertices will create triangles in the tetrahedral mesh. Optionally, normal vectors on each vertex can be calculated by interpolating the face normals, and given to the Mesh2 object, to eliminate sharp corners and give a smoother look to the object. POV-Ray generates bitmap files that contain the rendered scene for each frame of the animation. These bitmap files are then put together to create a movie of the animation, by using a movie editor software.

Chapter 5

Dust Formation

We implemented a simple addition to the system to support dust formation upon impacts. This way, we achieve better visual results when many small fragments are created as a result of breaking clay-like materials. The dust effect also offers favorable results when the scene contains objects falling on earth.

Dust formation is a process independent of the algorithm used for generating the fracture effects. If the user chooses to create dust for the animation, regardless of the practised algorithm, particle systems are created in case there is fracture. Each breaking object holds its own list of particle systems and is responsible for updating them. The number of generated particle systems depends on the number of impacts that trigger the fracture process. The particles are moved by basic particle dynamics, whereas the forces acting on the particles are calculated as a result of the impacts. Figure 5.1 presents selected frames from an animation in which dust is embedded. In the rest of this chapter, you can find the details of the particle system we use.

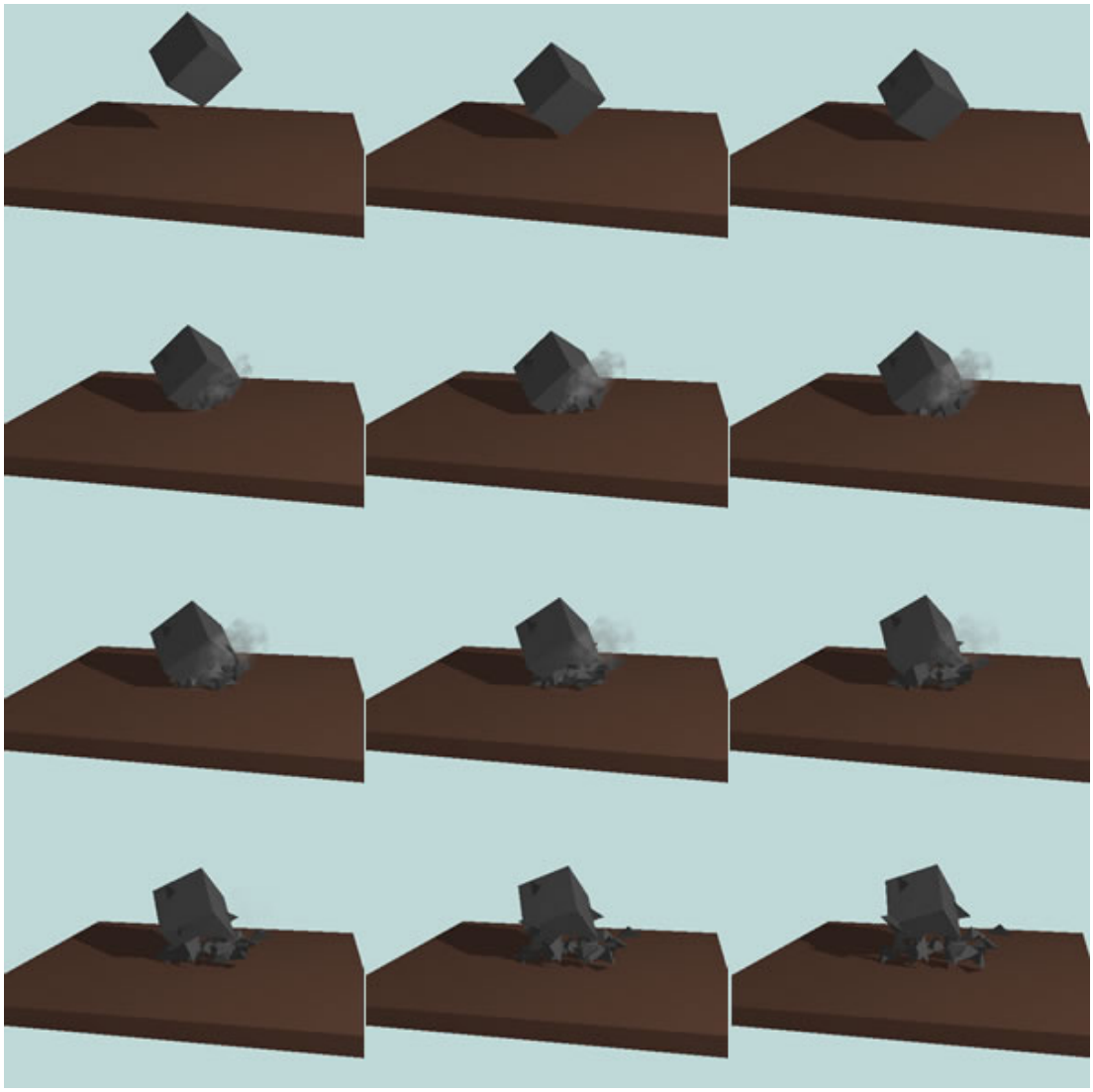

Figure 5.1: Dust formation upon impact

5.1 The Particle System

A particle system is basically a collection of points in space. These points, a.k.a. particles, go through a life cycle: they are born, change over time and finally die. Our particle system consists of particles and an emitter, which is responsible for creating the particles. Each particle in the system has position, velocity, and mass at a given time; it responds to forces and rests in the system for only a finite amount of time. Particles' behavior is controlled by a stochastic process. When a particle comes to the end of its period in the system, it is taken into a pool, from where newly added particles are taken randomly. The particle system we use is implemented as explained by Lander [9].

Witkin provides an excellent explanation of particle dynamics in [31]. The particle motion is governed by Newton's second law of motion:

$$F = ma \ . \tag{5.1}$$

This equation can be represented in the form of two equations,

$$\dot{v} = F/m \ , \tag{5.2}$$

$$\dot{x} = v \ . \tag{5.3}$$

Clearly, for a given particle, F denotes the sum of forces acting on it, m is the mass, a is the acceleration, v is the velocity, x is the position and finally t denotes time. Positions of the particles are updated at each animation frame by solving these two first order ODEs simultaneously.

The position and velocity, x and v can be concatenated to form a 6-vector, called *phase space*. The phase space equation of motion is

$$[\dot{x_1}, \dot{x_2}, \dot{x_3}, \dot{v_1}, \dot{v_2}, \dot{v_3}] = [v_1, v_2, v_3, F_1/m, F_2/m, F_3/m] \ . \tag{5.4}$$

A system of n particles are described by n copies of this equation, forming a $6n$-vector.

The second main element of the particle system is the emitter. The emitter is responsible for generating new particles at a given rate as long as the total particle count does not exceed a threshold. The particles are generated at random positions in a bounding volume. Additionally, the emitter can assign varying initial velocities to the particles for randomization effects.

The forces acting on each particle is calculated by a different module. These forces can be due to both force fields and forces caused by collisions with objects within the environment. The force fields can be constant force fields, such as gravity; time dependent force fields, such as wind or turbulence; and velocity dependent force fields, such as drag. Each particle in the system is represented by a structure containing its position, velocity (which alone make up the phase space), force, and mass. An Euler ODE solver is used to determine the motion.

Given a time step, an Euler solver takes the derivatives in equations 5.2 and 5.3, scales the derivative vectors with the time step, adds the current state vector of the particle with the newly calculated values, and finally iterates the particle system's clock.

In the implemented solution, particle systems are created at impact points whenever the impact magnitudes are over a threshold. An alternative solution to reduce the number of particle systems would be by grouping nearby impacts together and creating a single particle system for the whole group. The impact magnitude also effects the particle system's behavior. As the impact magniture gets bigger, the total number of particles hosted by the particle system increases, resulting with a denser dust formation. Also the global force acting on the particle system is a function of the linear momentum of the object and the impact magnitude. This way, the harder the object hits another, the farther the particles are scattered.

The particles in our animations stay in the system for a very short time for visual accuracy. Hence we omitted calculations for collision detection and

response. Since the particles in the system are large in number, it is infeasible to consider them as being scene objects. Normally, the particles are thought of as points and simple point-to-plane collision detection and response actions can be taken.

5.2 Rendering

Particles in the system are represented as spheres. Each sphere is assigned a media, which uses spherical mapping, where a color mapping is used to color the dust. The color map is a function, which accepts the distance to the center of the sphere as an input. Also a turbulence function is applied on the dust media. Without the turbulence, the view would be duller with every particle having the same color map.

Chapter 6

Experimental Results

6.1 Visual Results

Figure 6.1 illustrates selected frames from an animation generated by the slow algorithm. The scene contains two objects: a breakable ceramic bowl and the floor. The sequence is generated by dropping the bowl from an altitude. Hitting the floor, the model is broken into pieces. As it can be seen from the figures, the process generates many interlocking fragments, which obey the dynamics rules. The bowl model consists of 4514 tetrahedra. After the fracture, the parts of the bowl that are far away from the contact point remain intact. As explained in Chapter 4, this is a natural outcome of the first algorithm: the fracture is initiated at the crack region and it can propagate only in a local area.

Figure 6.2 shows a scene where a ball falls onto a glass table. The scene is created using the fast algorithm. The table consists of two rectangular legs and a glass surface sitting on the legs. The legs are fixed objects that are only supportive parts and are omitted during calculation. The only breakable object, table surface consists of 3581 tetrahedra. The animation is constructed by locating a non-breakable ball on top of the table, and assigning a linear momentum pointing downwards to that ball. As a result of that linear momentum, the ball falls onto the glass table. The impact at the collision is great enough to initiate fracture.

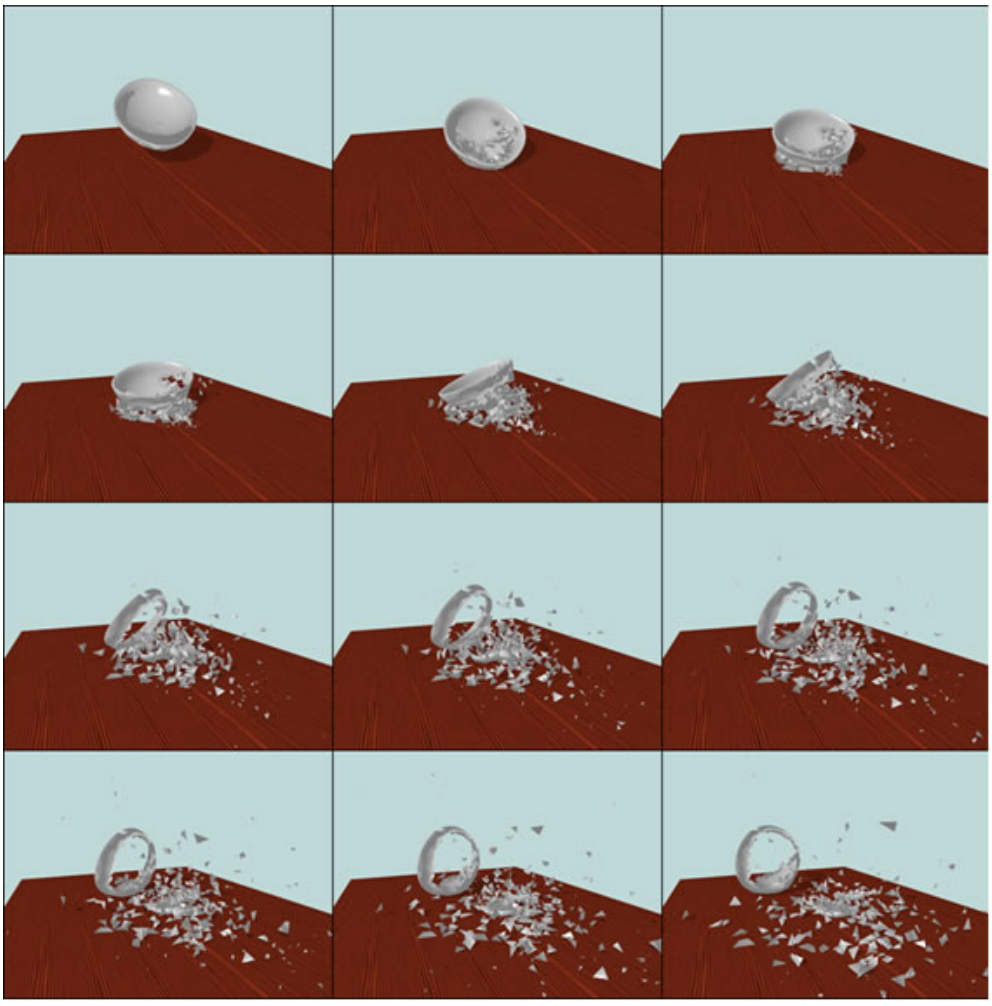

Figure 6.1: Ceramic bowl breaking upon falling to the ground (Animation generated by the slow algorithm)

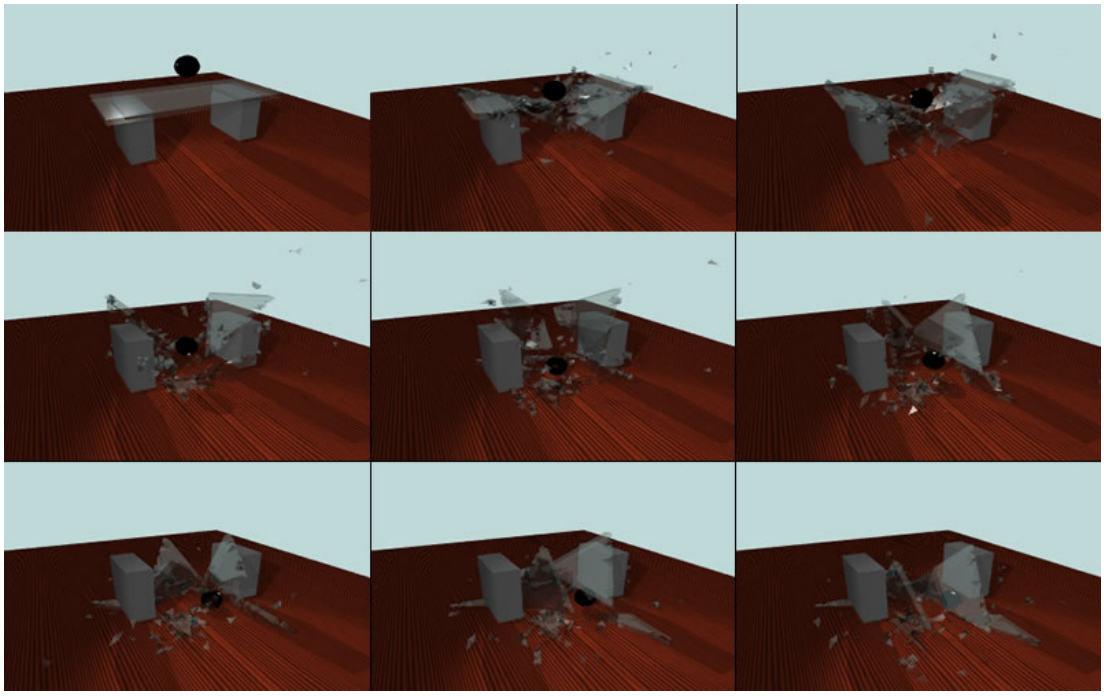

Figure 6.2: Glass table breaking under the impact of a heavy ball (Animation generated by the fast algorithm)

The fracture is propagated through planes and the body is fragmented into mainly four large pieces. Besides, numerous fragments are created, which begin to move naturally within the simulation scene. As a result of the changing geometry, the bigger fragments, which are still fragile, slide towards the floor. Yet they do not shatter as a result of the collision with the floor. This implies that the secondary collision does not arouse an impact strong enought to initiate fracture.

Figures 6.3 and 6.4 illustrate a visual comparison between the outputs of two algorithms. Both animations are created with exactly the same initial settings. The broken cube consists of 227 tetrahedra. In this comparison, the results of the second algorithm seem satisfying, even more realistic then those of the first algorithm. Note that in the output of the first algorithm (in Figure 6.3), we see a crack pattern only located near the impact region. This is expected since as we explained, the method uses a localized analysis when calculating the exerted forces on the bonds.

The output of the second algorithm (in Figure 6.4), on the other hand provides

a more diverse crack pattern. This pattern implies that the cube in our example was weaker at the diagonal. The tetrahedral meshes that NETGEN creates tend to be denser in the inner regions than they are at edges. As an assumption, we can say that the bigger the tetrahedra get, the strength of the bonds get weaker. Hence, we have a physically convincing result.

However, the second algorithm sometimes produce undesired results. In Figures 6.5 and 6.6, we see the resulting animation of the same scene, created with different algorithms. In this scene, the wall consists of 4080 tetrahedra. Again, in Figure 6.5 we see that the first algorithm only affects a small area on the wall. This is appropriate for a clay wall as rendered in the scene.

Figure 6.6 displays the output of the second algorithm. As mentioned earlier, this algorithm is dependent on the body properties of the object. Also since collision planes are used, the calculation of the collision radius is crucial. This calculation determines the crack growth range. Here in Figure 6.6, we see what happens when the collision radius is not selected properly. Such a distribution does not seem appropriate for a clay wall, but it would look better if the medium were glass. This implies that the second algorithm depends on some parameters which should be carefully selected by the animator. Selecting the radius differently, the animator might simulate a variety of different materials.

6.2 Performance Analysis

As discussed in Chapter 4, three major steps take place during the creation of an animation:

The first step, generation of the object models, is performed once for an animation scene, and the results are stored in a file. Also this is a very fast process, taking only a few seconds even for very large numbers of tetrahedra.

The second step, creation of the animation, is the most time consuming one. As seen in Figure 6.7, the generation of the frames that are created just as the

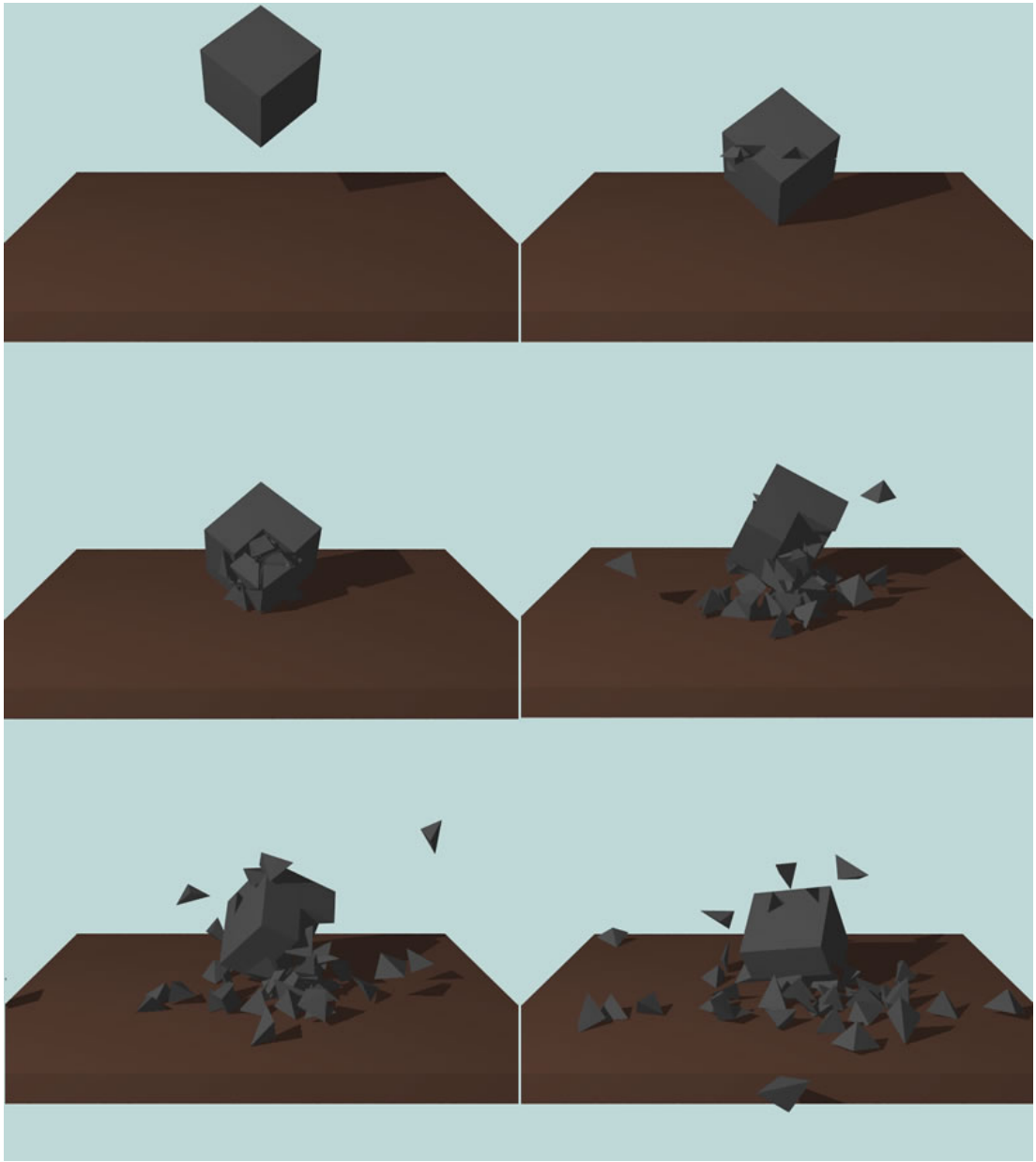

Figure 6.3: A cube is broken with the first algorithm

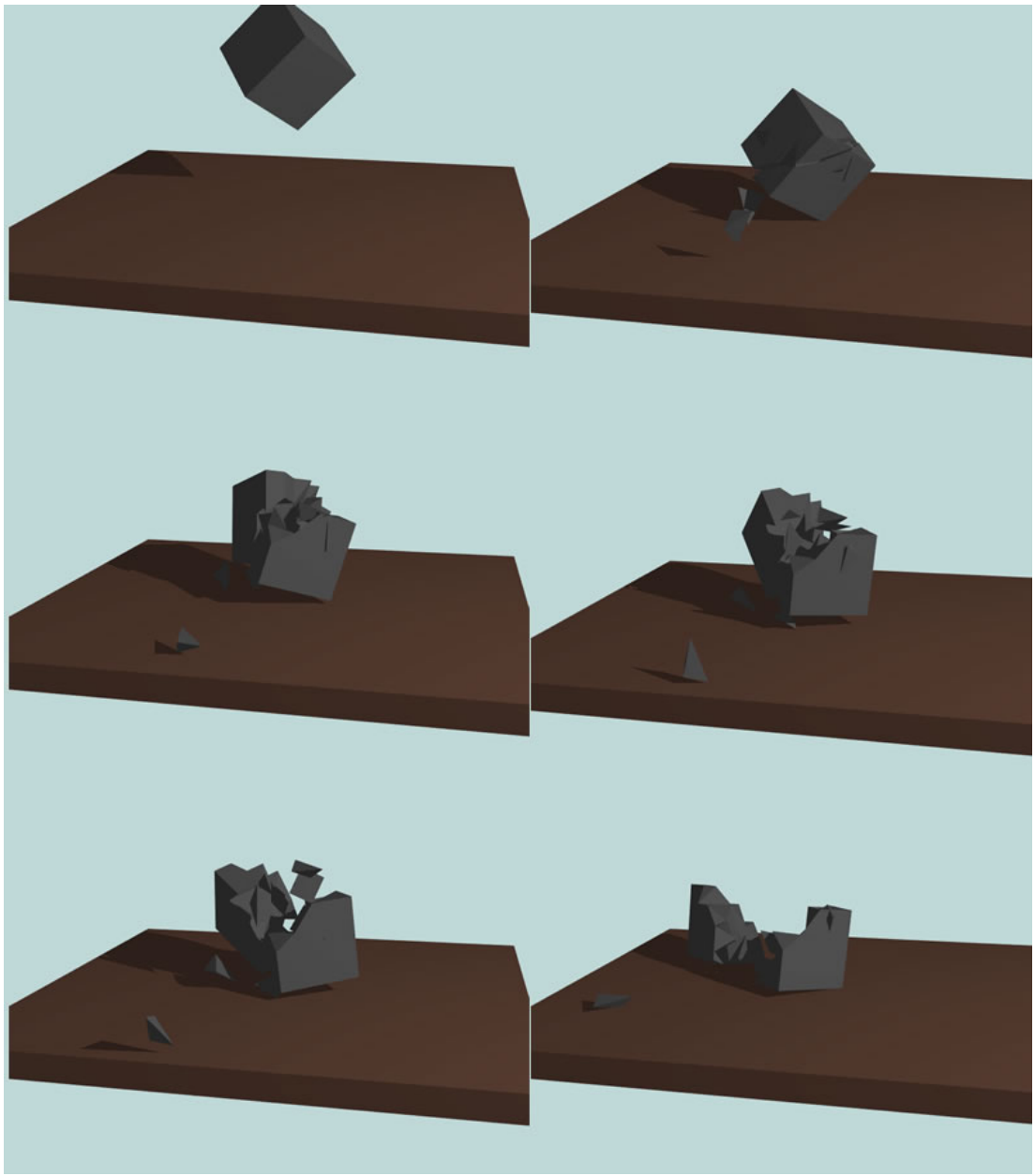

Figure 6.4: A cube is broken with the second algorithm

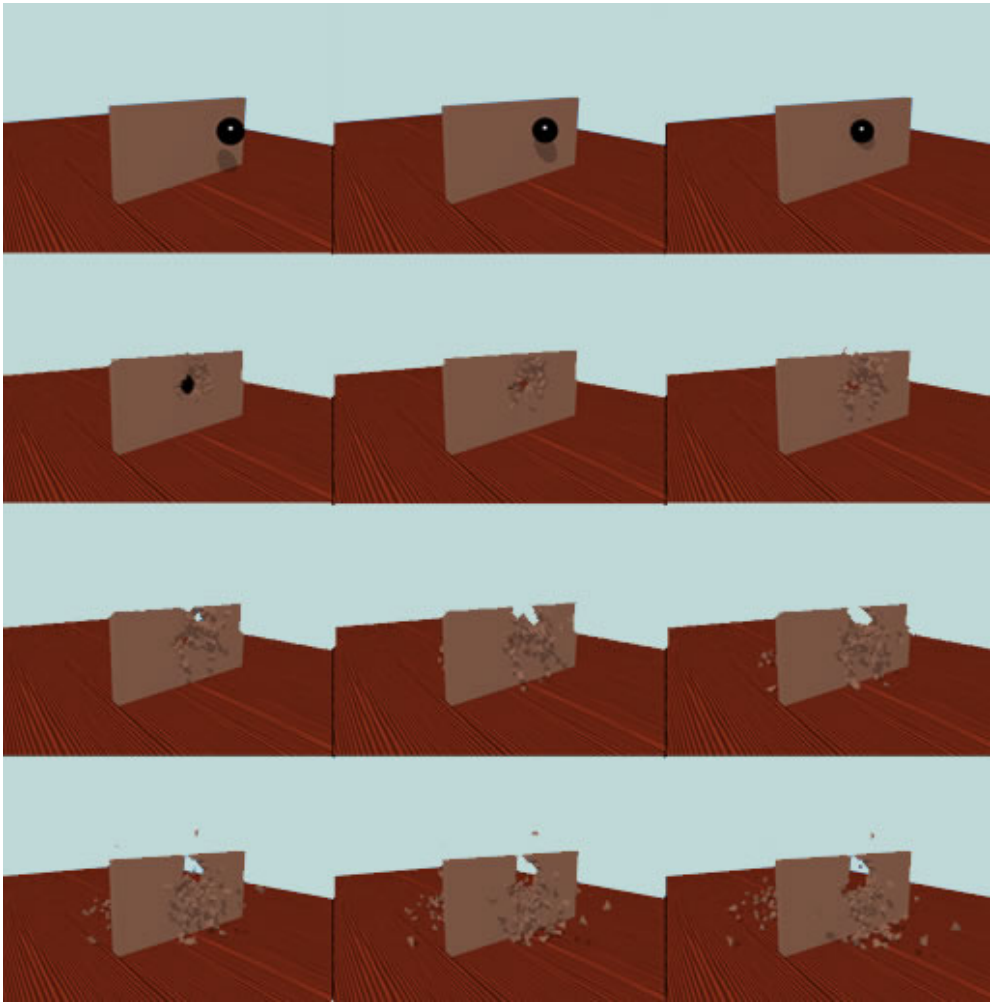

Figure 6.5: A clay wall is broken with the slow algorithm

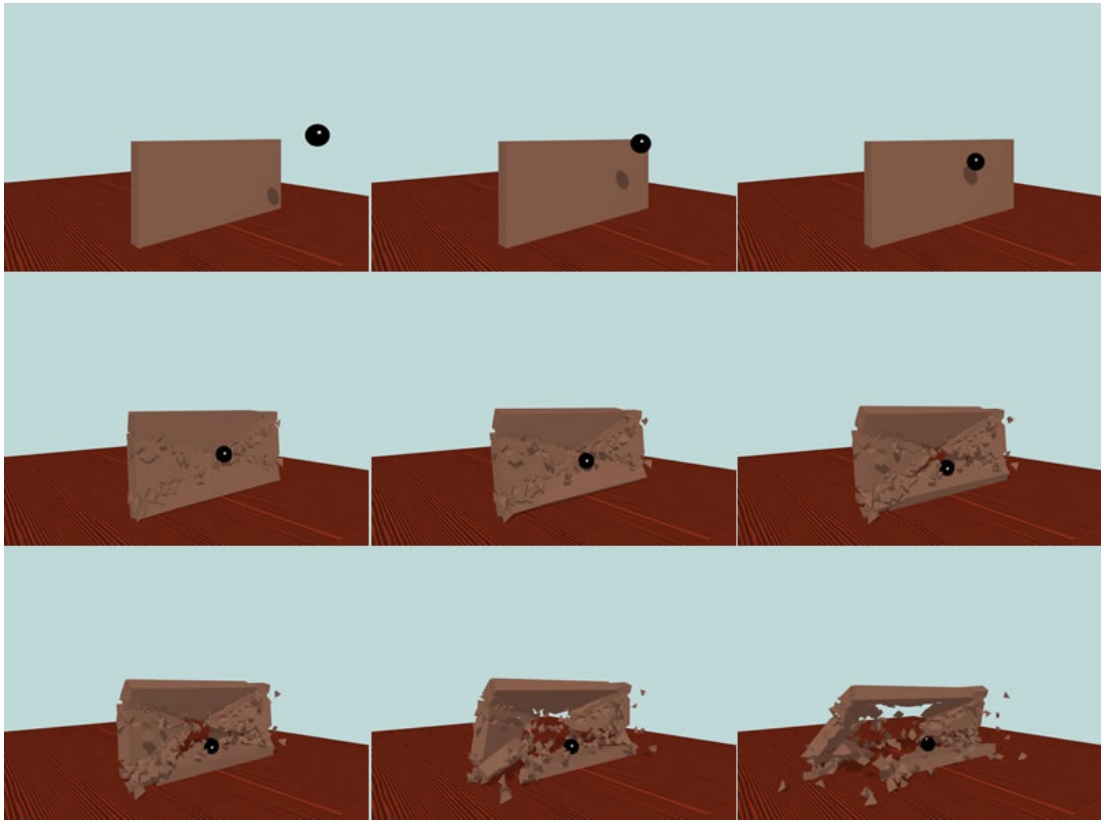

Figure 6.6: A clay wall is broken with the fast algorithm

shattering occurs take significantly more time than the remaining frames do. This bottleneck is solved significantly by the second algorithm. Note that even though this step remains to be very slow when compared with the rest of the simulation, there is no longer a huge gap between the processing time between frames.

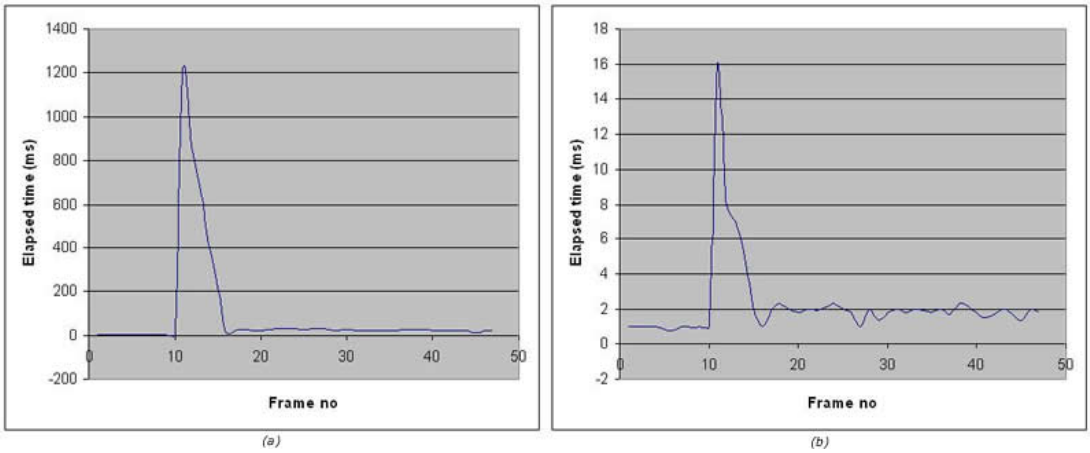

Figure 6.7: Calculation times for the breaking cube animation (a) - slow algorithm (b) - fast algorithm

In Table 6.1 the system's performance comparison between two methods is illustrated. The performance is considered as being the duration of the most time consuming step in the animation: the fracture calculations. In both cases, the performance is measured with respect to the number of impacts that initiate fracture and the number of tetrahedra that make up the broken body. It is obvious that as the contact count increases, the gain achieved by the second algorithm is significant even though the system cannot work in real-time. Also a comparison is provided for cases where the user selects initiating the dust creation process or not.

The time required for the third step, visualization of the results, depends on many parameters. Naturally the number of objects within a scene along with the material properties and complexity of these objects directly effects rendering performance. Scenes that contain materials prone to internal reflections, such as glass, take more time to render. The number of light sources can be a minor factor in case the orientation of the light source lets us percieve otherwise not apparent details of some objects. Experiments show that rendering dust affects

# of impacts	# of tetrahedra	Performance (in msec)			
		Fast		Slow	
		w dust	w/o dust	w dust	w/o dust
4	227	32	16	1235	1203
13	227	47	125	2265	2219
88	227	344	310	3031	2719
129	4080	812	329	128375	123937
220	4080	1625	531	246922	237063
967	4080	6688	1671	555250	472078

Table 6.1: Computation time of the fracture calculations versus impulse and tetrahedra counts for both algorithms

the performance unfavorably.

Chapter 7

Conclusion and Future Work

This study explores two approaches for animating brittle fracture in a realistic way. The implementation follows the methods explained in [23, 12]: The objects are initially represented as tetrahedral meshes. The tetrahedral mesh models are converted into lattice models, which are constructed by locating a point mass in the center of mass of each tetrahedron and connecting each neighboring mass couple with constraints. In our implementation, the constraint strengths can be further modified by some heuristics in order to simulate the irregularity in the material properties. For each animation frame, the objects' motion paths are calculated and the contacts between them are handled. For collisions involving breakable objects, the fracturing behavior of the object is determined by solving a system of dynamic constraints involving the tetrahedra that forms the object in the first method. The second method uses eigenvalue analysis and stress tensors for determining fracture.

Naturally, the number of the tetrahedra increases with respect to the complexity of the object geometry, and for generating nice looking animations. However, the time required for generating the animation increases as this number increases. Another limitation stems from the space requirements. The files describing the geometry of a high resolution tetrahedral object are quite large. As a result of this drawback, and considering the performances of the machines that were used for testing, the tetrahedron meshes were generated just as dense to illustrate

the breaking behavior. However, with high performance computers, much better animations could have been generated.

The animations generated by the formulation presented in section 4.3.1 outputs a fracture effect where there are several fragments consisting of single tetrahedron. Although, Smith *et al.* suggest in [12] that particles consisting of a single tetrahedron can be eliminated without loss of visual effects, this approach results with gaps around the cracks that seem to originate from nowhere. Therefore, in this study, single tetrahedron objects are left as is. However, this resulted with identical looking fragments, which can be seen in the results section.

Additionally, the constraint-based model is not sufficient on its own to mimic real world. As a result of the lattice construction, which assigns masses to tetrahedra according to their volumes, the density of objects stay uniform within the bodies. This imposes that an object is never weak at some parts of its body, or that any weakness is foreseen. This results in a uniform shattering effect, which seems dull. Since it is desirable to have irregularities in objects' mass distributions, some heuristics are applied in order to eliminate such uniformities by modifying the constraint strengths. Applying noise function on an object assigns different strengths to different parts of a body in a random manner, however, since the variation of the strengths is sparse, this heuristic does not produce appealing outputs. Alternatively, using a cleaving function, which modifies the strengths at given regions can be used. With cleaving, the animation can be controlled dynamically, by assigning strengths appropriately to regions that are desired to fall apart or stay intact.

The formulation presented in section 4.3.2 tries to solve some problems of the former solution. By calculating the stress over the whole body (or over some fraction of it), we achieve a global representation for material properties. In fact, the orientation and volumes of the tetrahedra composing the material gains importance. This way, longer cracks can be created on weaker parts of the object with less fragments. This helps enhancing the animations in some cases, but discrepancies in the lattice model make the fracture effect look artificial. Exploring the bodies more carefully, and setting material properties correctly is

crucial.

Also in this new formulation we have an important performance gain. Although the creation time of an animation depends on the number of impulses acting on the body, this is still an improvement over the former system, the performance of which depends not only on the number of impulses, but also on the number of tetrahedra a body is composed of.

At the moment a serious limitation of the system is its physical simulation engine. The physical movement of the bodies are calculated pretty efficiently when there is a small number of objects within a scene. Yet the performance decreases as soon as the number of objects increase. A solution to this problem would ease the system's load, letting it work faster.

As a result of the modeling, in both approaches, the fracture can occur only at the boundaries of the tetrahedra. An improvement to the current work, can be implementing local re-meshing, as explained by O'Brien and Hodgins [15]. This way, the mesh structure is recreated at the boundaries after the fracture, so that the cracks look smoother and more realistic. Also a totally distinct high-resolution surface mesh on top of a low resolution mesh can be used for rendering purposes only as in [11]. This would not affect the simulation performance, but might introduce a performance loss in updating the surface mesh.

Finally with the rendering software we have used, glass surfaces look unrealistic when there are several overlapping fragments, with respect to the viewpoint. Also, it was not possible to texture map a moving object, when the texture contains distinguishable patterns on it. Therefore, the animations produced and presented in the presented work demonstrate objects, the textures of which are either smooth or consist of tiny and almost identical patterns. Thus, the rendering part could be ameliorated to achieve better results.

Appendix A

Colliding Contact Derivations

In this appendix, we will thoroughly explain the derivations mentioned in Section 4.2.1, Colliding Contact Calculations. The formulas below are adapted from David Baraff's excellent course notes on rigid body simulation. For an alternative explanation, the reader can refer to [3].

In Section 4.2.1, v_{rel} that gives the component of the relative velocity $\dot{p_a}(t_c) - \dot{p_b}(t_c)$ in the $\hat{n}(t_c)$ direction is defined to be:

$$v_{rel} = \hat{n}(t_c)(\dot{p_a}(t_c) - \dot{p_b}(t_c)) \ . \tag{A.1}$$

The impulse that acts on objects that are in contact is:

$$I = \int F(t)dt \ , \tag{A.2}$$

If we apply an impulse I to a rigid body with mass m, then the change in linear velocity will be:

$$\Delta v = I/m \ , \tag{A.3}$$

making the change in linear momentum equal to

$$\Delta P(t_c) = I \ . \tag{A.4}$$

Similarly when impulse acts at point p, it produces an impulsive torque of

$$\tau_{impulse} = (p - x(t_c)) \times I \tag{A.5}$$

on the body with $x(t)$ being the center of mass of the object at time t, and t_c being the moment of the contact. Eventually the change in angular momentum equals impulsive torque $\tau_{impulse}$.

$$\Delta L(t_c) = (p - x(t_c)) \times I \ . \tag{A.6}$$

For a collision between the objects, the impulses applied on these objects are equal in magnitude but reverse in direction. Computing the impulse acting on an object, the impulse acting on the other object can be found simply by inverting the direction of the vector.

In case of a collision between bodies A and B, we shall denote the velocity of the contact point of body A, before the impulse I is applied, by $\dot{p}_a^-(t_c)$, and the velocity of the same point after the impulse is applied by $\dot{p}_a^+(t_c)$. $\dot{p}_b^-(t_c)$ and $\dot{p}_b^+(t_c)$ can be defined in a similar manner for body B. So, the relative velocities in normal direction before and after the application of I are:

$$v_{rel}^- = \hat{n}(t_c) \cdot (\dot{p}_a^-(t_c) - \dot{p}_b^-(t_c)) \ . \tag{A.7}$$

$$v_{rel}^+ = \hat{n}(t_c) \cdot (\dot{p}_a^+(t_c) - \dot{p}_b^+(t_c)) \ . \tag{A.8}$$

In frictionless systems, we have

$$v_{rel}^+ = -\epsilon v_{rel}^- \ , \tag{A.9}$$

where ϵ is called the *coefficient of restitution*, and defined in $[0, 1]$. This coefficient basically determines how *bouncy* the collision is.

In case of a collision, we calculate and apply the impulse to change the velocity of bodies immediately. In frictionless systems, the direction of the impulse will be in the normal direction, $\hat{n}t_c$. Thus, we can write I as

$$I = i\hat{n}(t_c) \ , \tag{A.10}$$

where i is the magnitude of the impulse and $\hat{n}(t_c)$ is the contact normal. What remains is to calculate the impulse magniture i.

After the impulse is applied, the velocity of the contact point on body A, which has post-impulsive linear velocity $v(t_c)$ and angular velocity $w(t_c)$ can be written as:

$$\dot{p_a^+}(t_c) = v_a^+(t_c) + w_a^+(t_c) \times r_a \ , \tag{A.11}$$

where r_a is the distance of the point to the center of mass of the object.

By A.3 linear velocity of body A after the impact is

$$v_a^+(t_c) = v_a^-(t_c) + \frac{i\hat{n}(t_c)}{M_a} \ . \tag{A.12}$$

Similarly, combining A.6 and 4.7 (in Chapter 4), angular velocity is defined as

$$w_a^+(t_c) = w_a^-(t_c) + l_a^{-1}(t_c)(r_a \times i\hat{n}(t_c)) \ , \tag{A.13}$$

where M_a is the mass of body A, and $l_a(t_c)$ is its inertia tensor as explained in chapter 4. Combining A.11, A.12, and A.13 we get

$$\begin{aligned}\dot{p}_a^+(t_c) &= \Big(v_a^-(t_c) + \tfrac{i\hat{n}(t_c)}{M_a}\Big) + \Big(w_a^-(t_c) + l_a^{-1}(t_c)\Big(r_a \times i\hat{n}(t_c)\Big)\Big) \times r_a \\ &= v_a^-(t_c) + w_a^-(t_c) \times r_a + \Big(\tfrac{i\hat{n}(t_c)}{M_a}\Big) + \Big(l_a^{-1}(t_c)\Big(r_a \times i\hat{n}(t_c)\Big)\Big) \times r_a \\ &= \dot{p}_a^-(t_c) + i\Big(\tfrac{\hat{n}(t_c)}{M_a} + l_a^{-1}(t_c)\Big(r_a \times \hat{n}(t_c)\Big)\Big) \times r_a \ .\end{aligned} \tag{A.14}$$

For body B applying an opposite impulse $-j$ yields

$$\dot{p}_b^+(t_c) = \dot{p}_b^-(t_c) - i\Big(\frac{\hat{n}(t_c)}{M_b} + l_b^{-1}(t_c)\Big(r_b \times \hat{n}(t_c)\Big)\Big) \times r_b \ . \tag{A.15}$$

Using these two equations, we get

$$\begin{aligned}\dot{p}_a^+(t_c) - \dot{p}_b^+(t_c) = \Big(\dot{p}_a^-(t_c) - \dot{p}_b^-(t_c)\Big) + i\Big(&\tfrac{\hat{n}(t_c)}{M_a} + \tfrac{\hat{n}(t_c)}{M_b} + \\ &\Big(l_a^{-1}(t_c)\Big(r_a \times \hat{n}(t_c)\Big)\Big) \times r_a + \\ &\Big(l_b^{-1}(t_c)\Big(r_b \times \hat{n}(t_c)\Big)\Big) \times r_b\Big) \ .\end{aligned} \tag{A.16}$$

By considering the two colliding objects A and B separately, we can define the relative velocity in the direction of the contact normal after the application of the impulse as:

$$\begin{aligned}v_{rel}^+ &= \hat{n}(t_c) \cdot \Big(\dot{p}_a^+(t_c) - \dot{p}_b^+(t_c)\Big) \\ &= \hat{n}(t_c) \cdot \Big(\dot{p}_a^-(t_c) - \dot{p}_b^-(t_c)\Big) + i\Big(\tfrac{1}{M_a} + \tfrac{1}{M_b} + \\ &\qquad \hat{n}(t_c) \cdot \Big(l_a^{-1}(t_c)\Big(r_a \times \hat{n}(t_c)\Big)\Big) \times r_a + \\ &\qquad \hat{n}(t_c) \cdot \Big(l_b^{-1}(t_c)\Big(r_b \times \hat{n}(t_c)\Big)\Big) \times r_b\Big) \ .\end{aligned} \tag{A.17}$$

Note that $\hat{n}(t_c) \cdot \hat{n}(t_c) = 1$ since $\hat{n}(t_c)$ is of unit length.

Since $\hat{n}(t_c) \cdot \left(\dot{p}_a^-(t_c) - \dot{p}_b^-(t_c)\right) = v_{rel}^-$, expressing v_{rel}^+ in terms of v_{rel}^- and combining this with A.9 we get

$$\begin{aligned} v_{rel}^- + i\Bigg(\tfrac{1}{M_a} + \tfrac{1}{M_b} + \hat{n}(t_c) \cdot \Big(l_a^{-1}(t_c)\Big(r_a \times \hat{n}(t_c)\Big)\Big) \times r_a + \\ \hat{n}(t_c) \cdot \Big(l_b^{-1}(t_c)\Big(r_b \times \hat{n}(t_c)\Big)\Big) \times r_b\Bigg) = -\epsilon v_{rel}^- \ . \end{aligned} \tag{A.18}$$

Finally solving for i, we get

$$i = \frac{-(1+\epsilon)v_{rel}^-}{\frac{1}{M_a} + \frac{1}{M_b} + \hat{n}(t_c) \cdot \Big(l_a^{-1}(t_c)\Big(r_a \times \hat{n}(t_c)\Big)\Big) \times r_a + \hat{n}(t_c) \cdot \Big(l_b^{-1}(t_c)\Big(r_b \times \hat{n}(t_c)\Big)\Big) \times r_b} \ . \tag{A.19}$$

Appendix B

Resting Contact Derivations

In this appendix, we will thoroughly explain the derivations mentioned in Section 4.2.2, Resting Contact Calculations. Again the reader can refer to [3], where these derivations are adapted from.

In case of colliding contacts we calculated an impulse $i\hat{n}(t)$ where the impulse magniture i was an unknown scalar. For resting contacts, we need to calculate the force $f\hat{n}(t)$ acting on each contact point. Let f_k be the magnitude of the force acting on k^{th} contact point and $\hat{n}_k(t)$ be the contact normal at that point. All f_k values should be computed simulateously since a force on a contact point can effect the bodies of another contact point.

It is necessary to enforce three constraint on contact forces. All the contact forces should prevent inter-penetration, they are repulsive and vanish as soon as the contact disappears. These can be formulated as follows:

For satisfying the inter-penetration constraint, we define $d_k(t)$ to describe the separation distance between the bodies near contact k at time t. $d_k(t) < 0$ implies inter-penetration, that is the bodies are pushed towards each other. At contact moment t_c, we will have $d_k(t) = 0$ and we need to make contact forces maintain $d_k(t) \geq 0, \forall t > t_c$.

In case of vertex-face and edge-edge contacts (see Figure 4.1), $d_k(t)$ is the

distance between the contact points $p_a(t)$ and $p_b(t)$ in the direction of contact normal $n_k(t)$:

$$d_k(t) = \hat{n}_k(t) \cdot (p_a(t) - p_b(t)) \ . \tag{B.1}$$

As explained above, if $d_k(t) > 0$, the bodies begin separating. Similarly if $d_k(t) = 0$, the bodies are just in contact. These two conditions are safe, but we need to prevent the case $d_k(t) < 0$. Hence we need to act early and keep $d_k(t)$ from decreasing further when it is zero. Taking the time derivative, we get:

$$\dot{d}_k(t) = \dot{\hat{n}}_k(t) \cdot (p_a(t) - p_b(t)) + \hat{n}_k(t) \cdot (\dot{p}_a(t) - \dot{p}_b(t)) \ . \tag{B.2}$$

As $d_k(t)$ denotes separation distance, $\dot{d}_k(t)$ denotes the separation velocity at time t. At $t = t_c$ we have $p_a(t_c) = p_b(t_c)$ which is the contact point. Hence we have $\dot{d}_k(t) = \hat{n}_k(t) \cdot (p_a(t) - p_b(t))$. We need to have $d_k(t_c) = \dot{d}_k(t_c) = 0$ in case of resting contacts. The bodies neither move towards or away from each other at contact time. Also for the separation acceleration, we have

$$\begin{aligned} \ddot{d}_k(t) = &\Big(\ddot{\hat{n}}_k(t) \cdot \Big(p_a(t) - p_b(t)\Big) + \dot{\hat{n}}_k(t) \cdot \Big(\dot{p}_a(t) - \dot{p}_b(t)\Big)\Big) + \\ &\Big(\dot{\hat{n}}_k(t) \cdot \Big(\dot{p}_a(t) - \dot{p}_b(t)\Big) + \hat{n}_k(t) \cdot \Big(\ddot{p}_a(t) - \ddot{p}_b(t)\Big)\Big) \\ = &\ \ddot{\hat{n}}_k(t) \cdot \Big(p_a(t) - p_b(t)\Big) + 2\dot{\hat{n}}_k(t) \cdot \Big(\dot{p}_a(t) - \dot{p}_b(t)\Big) + \\ &\ \hat{n}_k(t) \cdot \Big(\ddot{p}_a(t) - \ddot{p}_b(t)\Big) \ . \end{aligned} \tag{B.3}$$

Since we have $p_a(t_c) = p_b(t_c)$, at time t_c B.3 becomes

$$\ddot{d}_k(t_c) = \hat{n}_k(t_c) \cdot \Big(\ddot{p}_a(t_c) - \ddot{p}_b(t_c)\Big) + 2\dot{\hat{n}}_k(t_c) \cdot \Big(\dot{p}_a(t_c) - \dot{p}_b(t_c)\Big) \ . \tag{B.4}$$

Again, we should avoid $\ddot{d}_k(t_c) < 0$, since such a case indicates that the bodies are accelerating towards each other. Thus the inter-penetration constraint can be written as

$$\ddot{d}_k(t_c) \geq 0 \ . \tag{B.5}$$

For repulsiveness constraints, we must have all forces acting outward the bodies. Since $f_k\hat{n}_k(t_c)$ acts on body A where $\hat{n}_k(t_c)$ is the outwards pointing normal of B, we must have all f_k positive:

$$f_k \geq 0 \qquad \forall k \ . \tag{B.6}$$

For the last constraint, which is to enforce that the forces vanish as soon as the contact disappears, we have:

$$f_k\ddot{d}_k(t_c) = 0 \ . \tag{B.7}$$

With this, if the contact is breaking, the acceleration will be positive; hence the contact force will have to be zero in order to satisfy the constraint above. If the contact is not breaking, the acceleration will be zero; hence the constraint will be satisfied regardless of the value of the contact force as long as it is nonnegative.

In order to satisfy B.5, B.6, and B.7 we express $\ddot{d}_k$ as a function of yet unknown f_k's:

$$\ddot{d}_k(t_c) = \sum_{l=1}^{N} a_{kl} f_x + b_k \ , \tag{B.8}$$

where N is the number of contacts.

In matrix notation this can be restated as

$$\begin{pmatrix} \ddot{d}_1(t_c) \\ \vdots \\ \ddot{d}_N(t_c) \end{pmatrix} = \mathbf{A} \begin{pmatrix} f_1 \\ \vdots \\ f_N \end{pmatrix} + \begin{pmatrix} b_1 \\ \vdots \\ b_N \end{pmatrix} \ , \tag{B.9}$$

where $\mathbf{A}$ is the $N \times N$ matrix of a_{kl}'s.

If you are not interested in how the a_{kl} and b_k values are found, you can skip the rest of this appendix.

Assume that the k^{th} contact involves two bodies A and B. Let's restate B.3:

$$\ddot{d}_k(t_c) = \hat{n}_k(t_c) \cdot \Big(\ddot{p}_a(t_c) - \ddot{p}_b(t_c)\Big) + 2\dot{\hat{n}}_k(t_c) \cdot \Big(\dot{p}_a(t_c) - \dot{p}_b(t_c)\Big) \ .$$

The term $2\dot{\hat{n}}_k(t_c) \cdot \Big(\dot{p}_a(t_c) - \dot{p}_b(t_c)\Big)$ can be calculated immediately without knowing the forces, since this term is only velocity dependent. Hence this term contributes to the b_i values.

As mentioned earlier, a force on k^{th} contact point can effect the bodies of the l^{th} contact point. We are interested in how $\ddot{p}_a(t_c)$ and $\ddot{p}_b(t_c)$ effect f_l. If body A is not one of the bodies involved in l^{th} contact, then f_l does not act on body A, making $\ddot{p}_a(t_c)$ independent of f_l. A similar case holds for body B. However, if body A is involved in contact l, we need to derive how $\ddot{p}_a(t_c)$ is affected by the force acting on A. Assume that a force of $f_l\hat{n}_l(t_c)$ acts on A.

Let $p(t)$ be the world space coordinate, $R(t)$ be the orientation, and $x(t)$ is the position of the center of mass of body at time t. Also assume that the body space coordinate is p_0. Then,

$$p(t) = R(t)p_0 + x(t) \ . \tag{B.10}$$

We also have $r(t) = p(t) - x(t)$. This yields

$$\begin{aligned}
\dot{p}(t) &= \dot{R}(t)p_0 + \dot{x}(t) \\
&= w(t)^* R(t)p_0 + v(t) \\
&= w(t) \times \Big(R(t)p_0 + x(t) - x(t)\Big) + v(t) \\
&= w(t) \times \Big(p(t) - x(t)\Big) + v(t) \\
&= w(t) \times r(t) + v(t) \ .
\end{aligned} \tag{B.11}$$

Then the acceleration of the point is calculated as

$$\begin{aligned}\ddot{p}(t) &= \dot{w}(t) \times r(t) + w(t) \times \dot{r}(t) + \dot{v}(t) \\ &= \dot{w}(t) \times r(t) + w(t) \times \Big(w(t) \times r(t)\Big) + \dot{v}(t) \ .\end{aligned} \tag{B.12}$$

Here, the term $\dot{w}(t) \times r(t)$ is the acceleration perpendicular to the dispalcement $r(t)$. The second term $w(t) \times \Big(w(t) \times r(t)\Big)$ is the centripetal acceleration, which makes the points of body rotate in a circular orbit about the center of mass. Finally $\dot{v}(t)$ is the linear acceleration of the point.

For body A, B.12 can be rewritten:

$$\ddot{p_a}(t) = \dot{v_a}(t) + \dot{w_a}(t) \times r_a + w_a(t) \times \Big(w_a(t) \times r_a\Big) \ . \tag{B.13}$$

Since $\dot{v_a}(t)$ equals the total force acting on a body divided by the mass, $f_l \hat{n}_l(t_c)$ contributes to $\dot{v_a}(t)$ and also to $\ddot{p}_a(t)$:

$$\frac{f_l \hat{n}_l(t_c)}{M_a} = f_l \frac{\hat{n}_l(t_c)}{M_a} \tag{B.14}$$

Similarly we should consider angular acceleration, $\dot{w_a}(t)$ (see [3] for derivation):

$$\dot{w_a}(t) = l_a^{-1}(t)\tau_a(t) + l_a^{-1}(t)\Big(L_a(t) \times w_a(t)\Big) \ , \tag{B.15}$$

where $\tau_a(t)$ is the total torque acting on body A. The force exerted by the l^{th} contact, $f_l \hat{n}_l(t_c)$, creates a torque of

$$\Big(p_j - x_a(t_c)\Big) \times f_l \hat{n}_l(t_c) \ , \tag{B.16}$$

This way, the angular contribution to $\ddot{p}_a(t_c)$ becomes

$$f_l \Big(l_a^{-1}(t_c)\Big((p_j - x_a(t_c)) \times \hat{n}_l(t_c)\Big)\Big) \times r_a \ . \tag{B.17}$$

Combining B.14 with B.17, we get the total dependence of $\ddot{p}_a(t_c)$ on f_l:

$$f_l\left(\frac{\hat{n}_l(t_c)}{M_a} + \Big(l_a^{-1}(t_c)\Big((p_j - x_a(t_c)) \times \hat{n}_l(t_c)\Big)\Big) \times r_a\right) . \tag{B.18}$$

The depencende of $\ddot{p}_b(t_c)$ on f_l: is similar. In case the force acting on A were $-f_l\hat{n}_l(t_c)$, the total dependence would be the same, only with a different sign.

To compute the a_{kl}'s we combine $\ddot{p}_a(t_c)$'s and $\ddot{p}_b(t_c)$'s dependence on f_l together and take the dot product with $\hat{n}_k$.

At this point, we know the term $2\dot{\hat{n}}_k(t_c) \cdot \Big(\dot{p}_a(t_c) - \dot{p}_b(t_c)\Big)$ in B.3 contributes to b_i. We also need to take $\ddot{p}_a(t_c)$ and $\ddot{p}_a(t_c)$ into account since they contribute due to external forces and force-independent terms $w_a(t_c) \times \Big(w_a(t_c) \times r_a\Big)$ in B.13 and $l_a^{-1}(t_c)\Big(L_a(t_c) \times w_a(t_c)\Big) \times r_a$ in B.15. If total external force acting on A is $F_a(t_c)$ and the total external torque is $\tau(t_c)$ then the contributions of the force and torque respectively becomes:

$$\frac{F_a(t_c)}{M_a} \tag{B.19}$$

$$\Big(l_a^{-1}(t_c)\tau(t_c)\Big) \times r_a \quad . \tag{B.20}$$

Hence the part of $\ddot{p}_a(t_0)$ independent from all f_l's is

$$\begin{aligned} F_a(t_c)M_a + \Big(l_a^{-1}(t_c)\tau(t_c)\Big) \times r_a + w_a(t_c) \times \Big(w_a(t_c) \times r_a\Big) \\ + l_a^{-1}(t_c)\Big(L_a(t_c) \times w_a(t_c)\Big) \times r_a \ . \end{aligned} \tag{B.21}$$

The constant part for $\ddot{p}_b(t_0)$ is computed similarly. In order to get b_k, we combine both constant parts for A and B, dot the term with $\hat{n}_i(t_c)$ and add the term $2\dot{\hat{n}}_k(t_c) \cdot \Big(\dot{p}_a(t_c) - \dot{p}_b(t_c)\Big)$.

Appendix C

The System at Work

The system is implemented in C++ using Microsoft® Visual C++® .NET Development Environment 2003 with OpenGL® graphics API. The user interface is developed using Microsoft® Win32® API[1].

The user interface is simple and mainly lets the user create dynamic scenes, edit properties of the objects in the scene, and run the simulation frame by frame.

Upon launching, two windows are created. *Simulation Window* displays the scene and contains the menu bar. Initially an empty scene is opened within the Simulation Window (see Figure C.1). *Output Window*, which can be toggled on and off, displays important information such as loading a scene, collision data, finished simulation steps etc. Actually any textual data written on this screen is also written to a log file during the simulation, yet this window visually enhances the system's usability.

File menu, shown in Figure C.2, contains utilities to help the user create a scene. The user can open a previously saved scene from the hard drive using the *Open Scene* dialog, or create a brand new scene by selecting objects via *Import Object* dialog. At any point the user can clear the scene with the *New Scene* option. Naturally any scene can be saved by clicking *Save Scene*. The scene files

[1]Microsoft, Visual C++, and Win 32 are registered trademarks of Microsoft Corporation in the U.S.A. and/or other countries. OpenGL is a registered trademark of SGI.

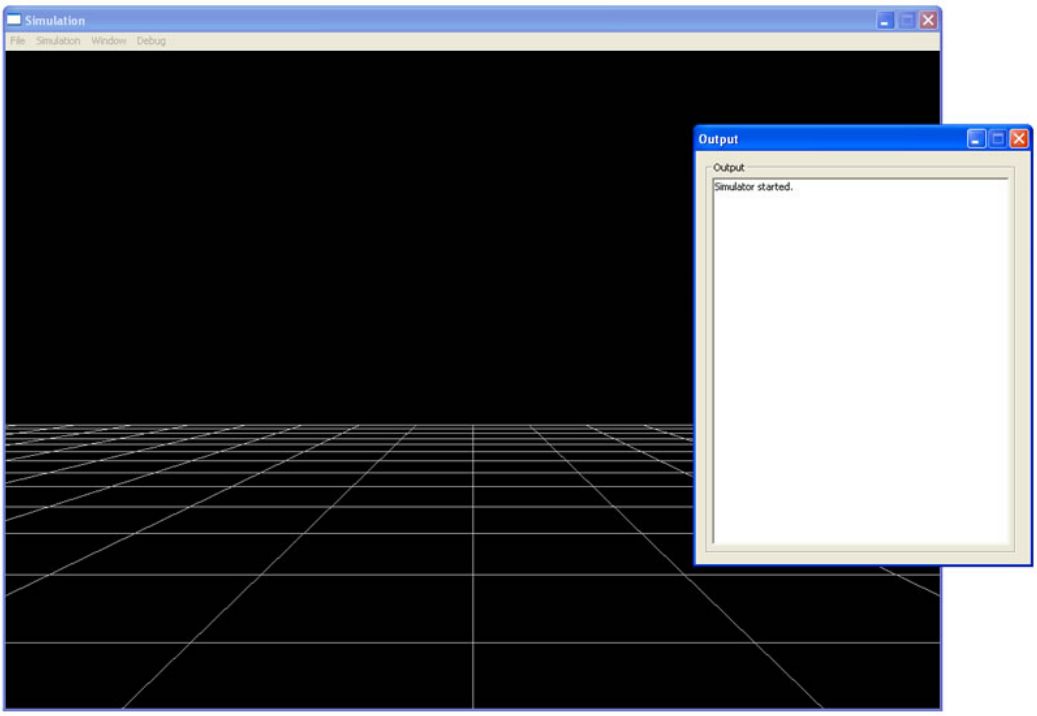

Figure C.1: Main screen

have *.scn* extension. A scene file contains the number of objects in a scene and scene-related data such as the world coordinates, orientations, momentums of the objects. Object data is read from two separate files. Tetrahedral mesh data is stored in *.tri* files. A tetrahedral mesh data file contains the coordinates of the vertices of an object as well as the vertex indices that make up the faces. Lattice data is held in *.lat* files. A Lattice data file holds the positions and masses of the lattice points, the indices of the neighboring lattice points, and connection strengths for each of these neighbors. When the user chooses to open a scene, the system reads data from all these files.

On the other hand importing a single object into a scene, the system reads a Netgen *.node* file. This file is very similar to the system generated *.tri* files with minor modifications. Since no lattice data is included in these files, the system generates the lattice and creates a body for simulation.

Figure C.3 shows a scene with two objects. The objects are rendered without regard to their visual qualities since the purpose of the system is to show the user

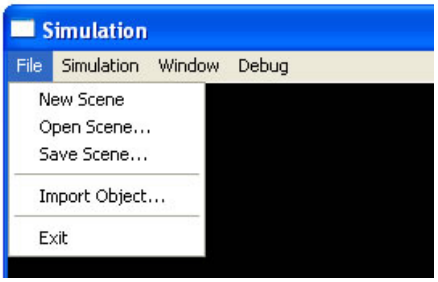

Figure C.2: File Menu

how the animation will develop. The final rendering is done later offline.

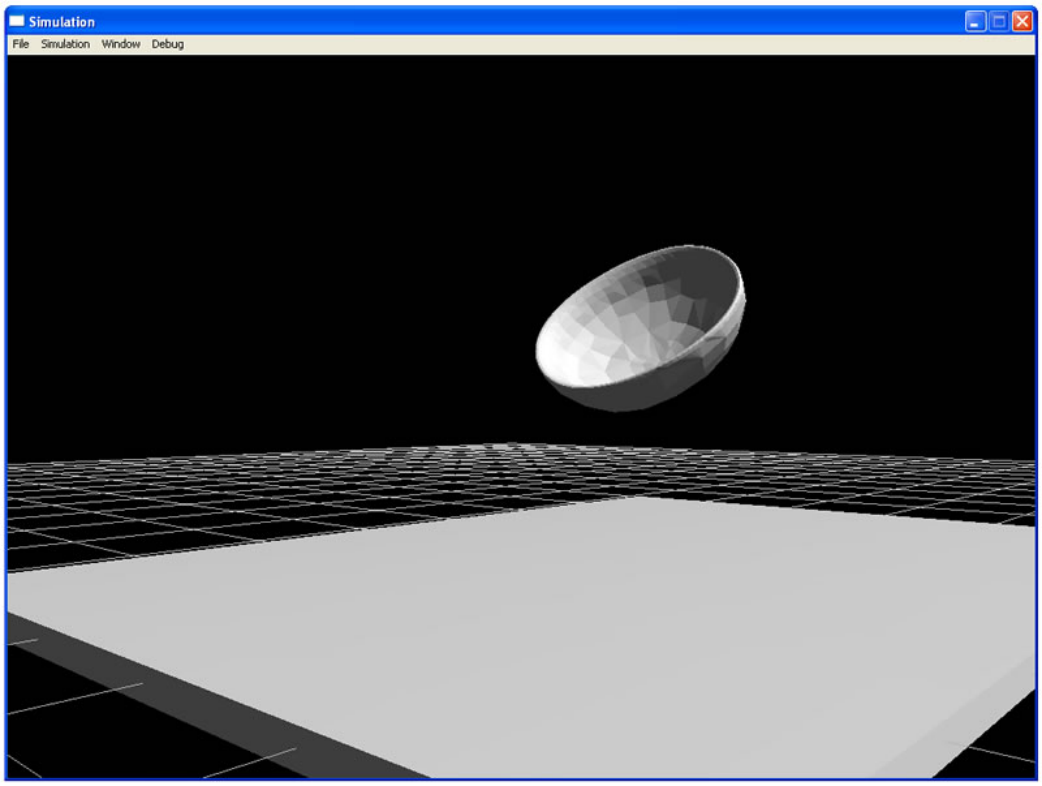

Figure C.3: A sample scene

The user can now wander around the scene using the direction keys. Also he/she can select certain objects to edit desired properties. Whenever an object is selected, its mesh is highlighted in red so that it is visually perceivable which object is selected as seen in Figure C.4.

Simulation menu (C.5) hosts most of the functionality the system provides.

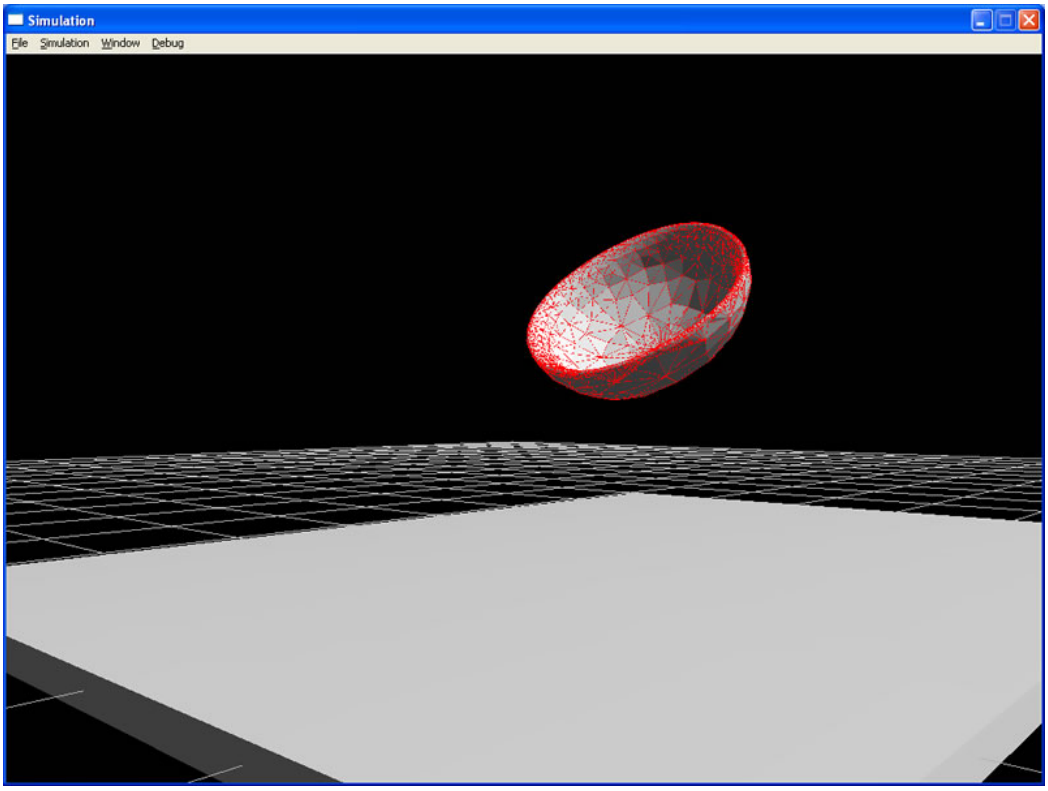

Figure C.4: A selected object highlighted with red

The first group in this menu contains tools for iterating through a simulation. As the name implies *Restart* loads back to the initial state of a scene. *Step forward* option lets the user advance to the next animation frame. In case the user wants to automatically go to a future frame without the need for selecting the *Step forward* at each frame, he/she can select the *Go to Frame* option. This option opens a dialog which prompts for the frame number to advance to (C.6). In case the number of a former frame is entered, the system does nothing.

The next group in *Simulation menu* helps editing the object properties of a selected object. *Edit Initial Body Properties* and *Edit Current Body Properties* options are disabled if no object is selected. Selecting one of these options open the *Object Properties* dialog in Figure C.7. This dialog lets the user edit all object data. User can change the mass, size, or texture of the object; give it a name; set static properties such as its being fixed, breakable, smooth, or stable; and assign a new position, orientation or momentum. The difference between the two

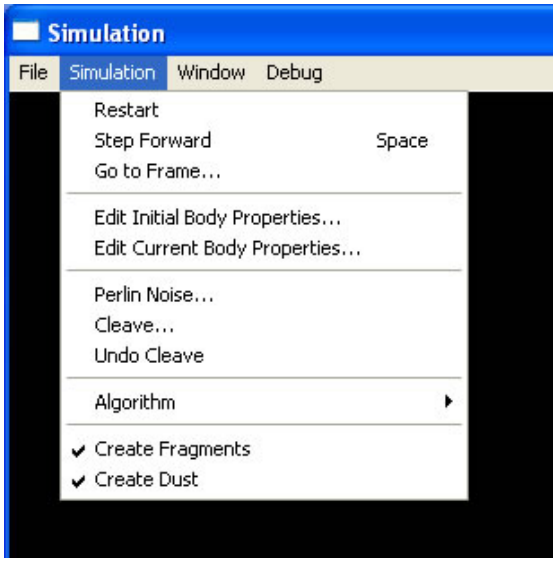

Figure C.5: Simulation Menu

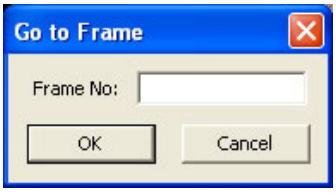

Figure C.6: Go to Frame Dialog

options are minor. Editing the initial body properties might come handy if the user wants to restart the animation many times with a different configuration without the need to save the scene. The changes made with this option has no effect on the ongoing simulation. However, if the user wants to observe the changes immediately, he/she should use the second option.

The third group within this menu contains tools for changing the objects' connection strengths by hand. As explained in Chapter 3, these tools are applying procedural Perlin noise functions and using cleaving planes. Selecting each option opens the respective dialog shown in Figures C.8 and C.9.

The meaning of the α, β, and n parameters for the Perlin noise function can be found in [18, 19]. The *Cleaving dialog* asks for two planes where a plane is represented by a point on the plane and a normal -hence the x, y, and z stand for the coordinates of these vectors-, and the amount of strength that will be applied

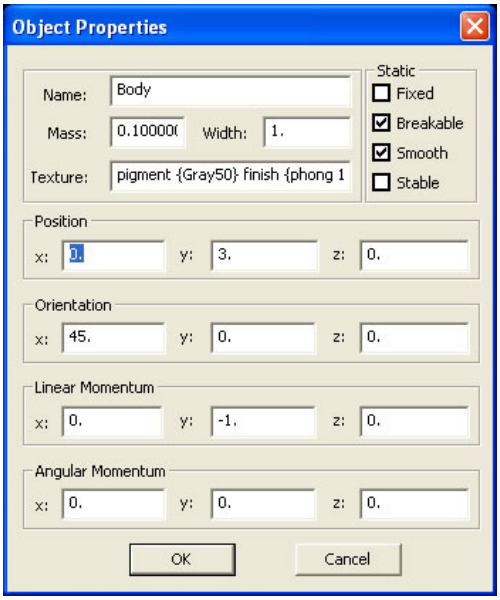

Figure C.7: Object Properties Dialog

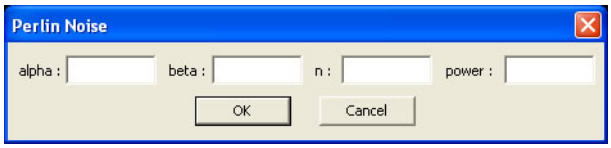

Figure C.8: Apply Perlin Noise Dialog

on the connections falling between these two planes.

Undo Cleave option restores the object to the state that the effect of the last cleaving applied on it is discarded. Unfortunately no such option is currently available for undoing the noise effect.

The *Algorithm* option within *Simulation menu* opens a sub-menu in which the user chooses the algorithm he/she wants to run for realizing the fracture (Figure C.10). The algorithms are named *Stress tensor based* and *Lagrange multiplier based*, standing for the fast and the slow algorithms respectively. Near each algorithm, its speed is indicated in qualitative terms within parantheses.

The last group in this menu provides the user to select some extra features. As explained in Chapter 4, the fragments created upon fracturing of an object inherits that object's properties. Hence these fragments, too, are subject to

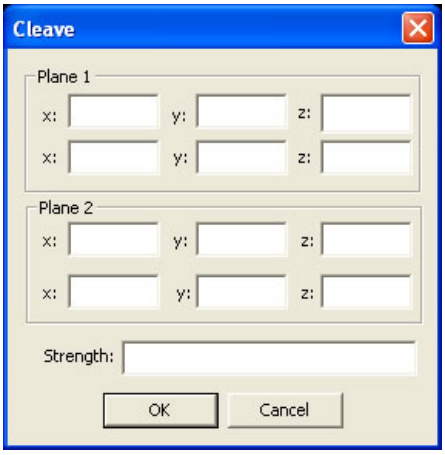

Figure C.9: Use Cleaving Planes Dialog

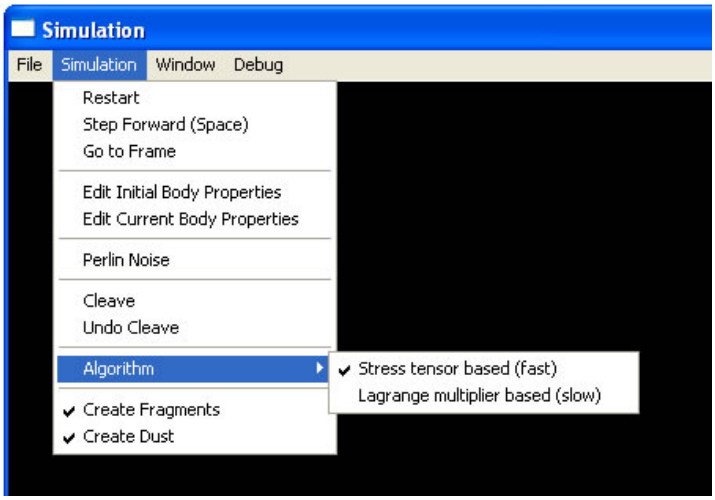

Figure C.10: Algorithm choices within the Simulation menu

breaking. However, the user may want to create a single crack and stop the fracture process for the child objects. In this case, all he/she needs to do is to uncheck the *Create Fragments* option. By default this option is checked.

Creating dust, as explained in Chapter 5, can be realized if the user checks the *Create Dust* option at any time of the simulation before the impact is eventuated. By default this option is unchecked.

Window menu (Figure C.11) is responsible for enclosing the functions related with the visual properties of the windows. *Show Output* option opens and closes the Output window shown in Figure C.1. *Render Surface* is an option on how

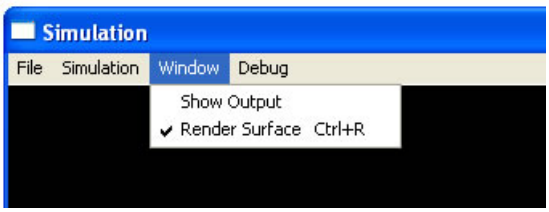

Figure C.11: Window Menu

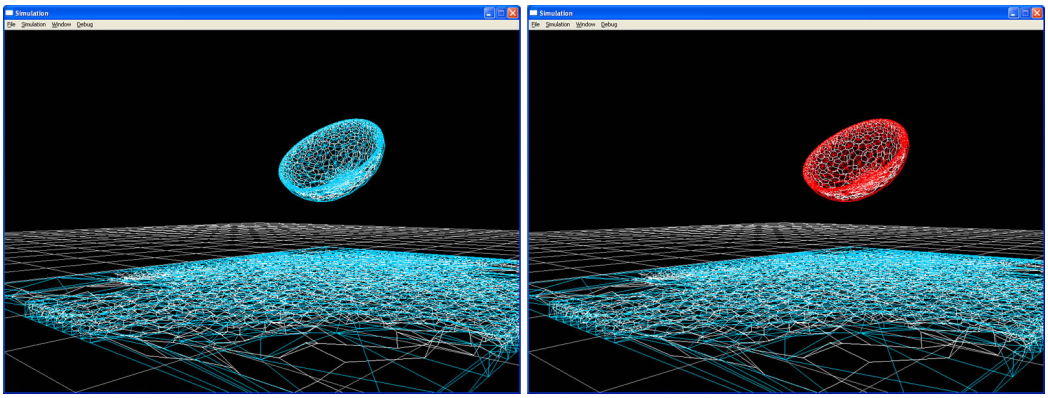

Figure C.12: A Scene where Objects are Rendered as Meshes

to see the objects within a scene. In case the user wants to visualize the objects as meshes, he/she can uncheck this option. Then the scene will look as in Figure C.12. In this mode, the user can still observe the selected object highlighted in red. Both the tetrahedral mesh and the lattice are rendered in this mode. The tetrahedral mesh is indicated in light blue whereas the lattice will be regionally colored according to the strength of the connections. The lattice is colored white by default. If the user applies cleaving or perlin noise on the initial model, the connection strenths change. The lines in rendered lattice stand for connections in the model. If these connections get stronger upon a modification, they are marked via a darker blue tone. The bluer the line, the stronger the connection. On the other hand, if the connections are weakened, they are colored in shades of red. Figure C.13 shows a sample object after applying a noise function on it.

Finally, *Debug menu* contains a single option to output object data to a text file of the user's choice. This file is simply a combination of the information in tetrahedral mesh data file and lattice file with the objects scene-dependent properties. This file is more understandable than the ordinary data files, since it

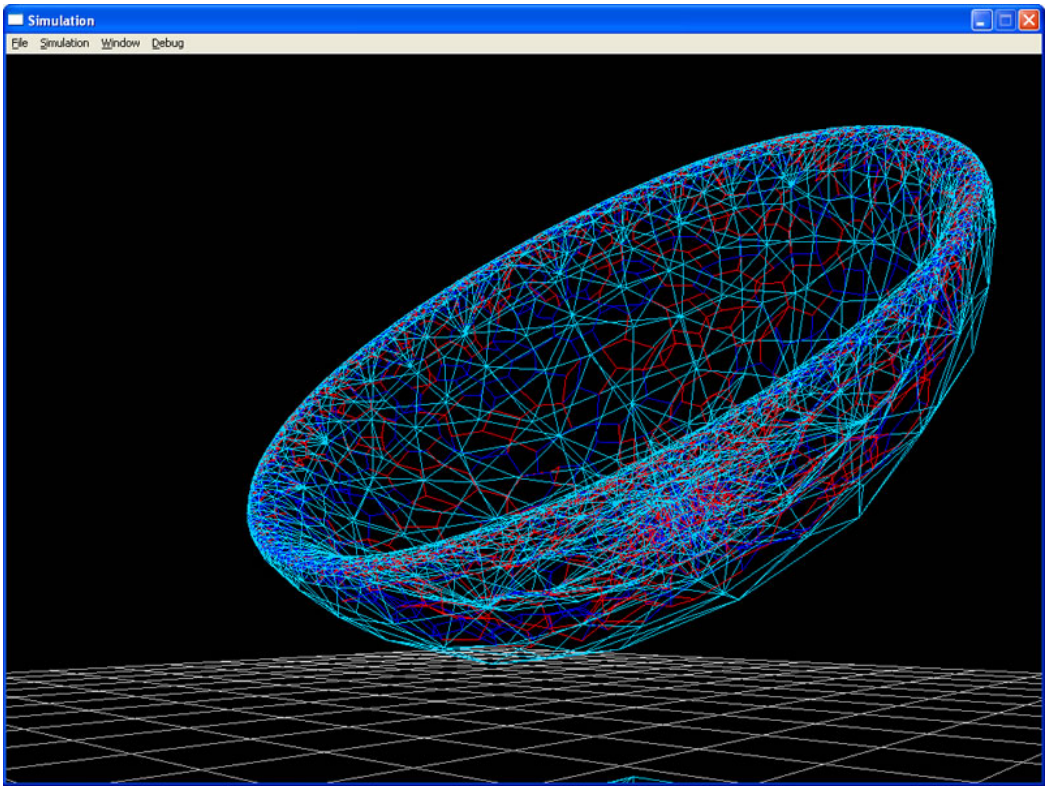

Figure C.13: A sample coloring scheme of the lattice after applying noise function on the object

contains additional user-friendly comments.

Bibliography

[1] D. Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. *SIGGRAPH Comput. Graph.*, 23(3):223–232, 1989.

[2] D. Baraff. Non-penetrating rigid body simulation. In *State of the Art Reports, Eurographics '93*, Sept. 1993.

[3] D. Baraff. *Rigid Body Simulation*, volume 21 of *ACM Siggraph 2001, Course Notes*, pages G1–G68. ACM Siggraph, New York, USA, 2001.

[4] C. Colefax. The persistence of vision raytracer [web page]. `http://www.povray.org/`, 2007.

[5] B. Desbenoit, E. Galin, and S. Akkouche. Modeling cracks and fractures. *The Visual Computer*, 21(8-10):717–726, 2005.

[6] E. M. Gertz and S. J. Wright. Object-oriented software for quadratic programming. *ACM Trans. Math. Softw.*, 29(1):58–81, 2003.

[7] E. M. Gertz and S. J. Wright. OOQP: Object-oriented software for quadratic programming [web page]. `http://pages.cs.wisc.edu/~swright/ooqp`, 2007.

[8] S. Gibson and B. Mirtich. A survey of deformable modeling in computer graphics. Technical Report TR-97-19, Mitsubishi Electric Research Lab, 1997.

[9] J. Lander. The ocean spray in your face. *Game Developer*, 5(7):13–19, July 1998.

[10] A. Martinet, E. Galin, B. Desbenoit, and S. Hakkouche. Procedural modeling of cracks and fractures. In *SMI '04: Proceedings of the Shape Modeling International 2004 (SMI'04)*, pages 346–349, Washington, DC, USA, 2004. IEEE Computer Society.

[11] M. Müller and M. Gross. Interactive virtual materials. In *GI '04: Proceedings of Graphics Interface 2004*, pages 239–246, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.

[12] M. Müller, L. McMillan, J. Dorsey, and R. Jagnow. Real-time simulation of deformation and fracture of stiff materials. In *Proceedings of the Eurographic workshop on Computer animation and simulation*, pages 113–124, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[13] A. Norton, G. Turk, B. Bacon, J. Gerth, and P. Sweeney. Animation of fracture by physical modeling. *Vis. Comput.*, 7(4):210–219, 1991.

[14] J. F. O'Brien, A. W. Bargteil, and J. K. Hodgins. Graphical modeling and animation of ductile fracture. *ACM Transactions on Graphics (SIGGRAPH 2002)*, 21(3):291–294, July 2002.

[15] J. F. O'Brien and J. K. Hodgins. Graphical modeling and animation of brittle fracture. In *Proceedings of SIGGRAPH 99*, pages 137–146, Aug. 1999.

[16] J. F. O'Brien and J. K. Hodgins. Animating fracture. *Communications of the ACM*, 43(7):68–75, 2000.

[17] M. Pauly, R. Keiser, B. Adams, P. Dutré, M. Gross, and L. J. Guibas. Meshless animation of fracturing solids. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 957–964, New York, NY, USA, 2005. ACM Press.

[18] K. Perlin. An image synthesizer. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296, New York, NY, USA, 1985. ACM Press.

[19] K. Perlin. Improving noise. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682, New York, NY, USA, 2002. ACM Press.

[20] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[21] J. Shcöberl. NETGEN - v4.3 [web page]. `http://www.hpfem.jku.at/netgen/`, 2003.

[22] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[23] J. W. Smith, A. Witkin, and D. Baraff. Fast and controllable simulation of the shattering of brittle objects. In *Proceedings of Graphics Interface 2000*, pages 27–34. Lawrence Erlbaum Associates, 2000.

[24] J. W. Smith, A. Witkin, and D. Baraff. Fast and controllable simulation of the shattering of brittle objects. *Computer Graphics Forum*, 20(2):81–91, 2001.

[25] D. Terzopoulos and K. Fleischer. Modeling inelastic deformation: viscoelasticity, plasticity, fracture. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 269–278, New York, NY, USA, 1988. ACM Press.

[26] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer. Elastically deformable models. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 205–214, New York, NY, USA, 1987. ACM Press.

[27] D. Terzopoulos and A. Witkin. Physically based models with rigid and deformable components. *IEEE Comput. Graph. Appl.*, 8(6):41–51, 1988.

[28] G. van den Bergen and R. Hernandez. FreeSOLID collision detection library - v2.1.1 [web page]. `http://sourceforge.net/projects/freesolid/`, 2007.

[29] Wikipedia. Stress (physics) [web page]. `http://en.wikipedia.org/wiki/Stress_%28physics%29`, 2007.

[30] A. Witkin. *Differential Equation Basics*, volume 21 of *ACM Siggraph 2001, Course Notes*, pages B1–B8. ACM Siggraph, New York, USA, 2001.

[31] A. Witkin and D. Baraff. *Particle Dynamic*, volume 21 of *ACM Siggraph 2001, Course Notes*, pages G1–G68. ACM Siggraph, New York, USA, 2001.